DESIGN AND DEVELOPMENT OF COST-EFFECTIVE COMPUTER INTERFACING SYSTEMS FOR MATHEMATICAL PROGRAMMING ALGORITHMS

A THESIS

Presented to

The Faculty of The Division

of Graduate Studies

By

Christos Paul Papacostadopoulos

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Operations Research

Georgia Institute of Technology

August, 1977

DESIGN AND DEVELOPMENT OF COST-EFFECTIVE COMPUTER INTERFACING SYSTEMS FOR MATHEMATICAL PROGRAMMING ALGORITHMS

Approved:

Dr. J.J. Jarvis

Dr. P.H. Enslow

Dr. D.B. Young

Dr. M.V. Konopasek

6/15/77

Date Approved by Chairman

I gratefully dedicate this thesis to my parents, Paul and Irene.

ACKNOWLEDGMENTS

I am very grateful to Dr. J. J. Jarvis for introducing me in this area and for his guidance and encouragement in the preparation of this thesis. His many helpful ideas and suggestions made this thesis possible. His patience and kindness are indeed appreciated.

I would like to thank Dr. P. Enslow, Dr. M. Konopasek and Dr. D. Young for reading the thesis. Their ideas and suggestions were very valuable and their contribution to this thesis is very much appreciated.

I also wish to thank Dr. R. Demillo and Dr. C. Park for their suggestions and comments. Special thanks should go to my fellow graduate student, Frank Cullen, for his interest, encouragement and helpful suggestions.

I would like to express my gratitude to the National Science Foundation for supporting the research in the area of interactive linear programming (Grant No. SED 75-17476A01). This research constitutes a significant portion of this thesis. I also would like to acknowledge the School of Textile Engineering for the opportunity to participate on research projects indirectly related to the topic of my thesis and for providing support during the last stage of my studies.

Mrs. Sharon Butler is an excellent typist and I thank her for her services.

TABLE OF CONTENTS

Chapter Page

LIST OF TABLES

LIST OF ILLUSTRATIONS

SUMMARY

The research concentrated on the area of effective man-machine communication for mathematical programming. Three directions of investigation were considered. First, features that the interfacing system should provide to the user. Second, how should the communication language be recognized and analyzed effectively? Third, how the characteristics of the man-machine system affect the long run effectiveness of the overall system.

With regard to the first direction of research, the result was that the features of the system should elevate the level of communication closer to the problem environment. This could be accomplished by high input flexibility, high rate of communication and provisions for utilizing human experience and judgment.

With regard to the second direction of research, it was shown that the languages for operations research procedures can be generated by context-free grammars. The theory of languages and automaton can be applied for the analysis and recognition of these languages. Two approaches for recognition and analysis were presented: a formal approach based on the operation of automata and a graphical approach. The latter is a state graph representation of the grammar and the recognition process is a flow through various syntactical states.

The thesis indicated how various syntactical features affect the complexity of the language and the recognition cost. An overview of data structures with respect to mathematical programming was given.

With regard to the third direction of research, economic analysis and user psychology of man-machine systems were investigated. A model for cost-benefit analysis was proposed. The model evaluates the present worth of the system over an infinite time horizon. Psychological considerations were investigated and it was stressed that studying and understanding human behavior is an important part of the design of man-machine systems.

CHAPTER I

INTRODUCTION

I.1. Introduction

Operations research has evolved into a science of a high degree of sophistication. Algorithms have been developed for solving a variety of problems. Many of these algorithms are recognized for efficiency and universal application. The evolution paralleled the growth of computer science and technology, since most operations research methods require a digital computer for efficient cost-effective utilization.

There is, however, a difficulty that causes most of these sophisticated computer algorithms to be underutilized, the barrier associated with communication between man and machine. Access to these algorithms has required considerable computer skills and special knowledge of program structure. Potential users tend to avoid sophisticated computer codes for operations research methods.

I.2. Purpose of the Thesis

The purpose of this thesis is to develop and demonstrate the design of interface systems for cost-effective communication of the user with computerized operations research procedures. These interfacing systems should provide trade-offs between costs incurred by the user and the machine. In

particular, the research will be in the area of interfacing methods for implementing optimization techniques known as mathematical programming. Emphasis will be given to flexibility of accessing the algorithm, cost-effectiveness, and pedogogical or self teaching aspects of the interface systems. Single pass and multipass methods which take advantage of the characteristics of the input model, for soliciting, storing and analyzing the input information, will be investigated. Concepts in higher level interfacing systems will also be explored.

Linear programming will be the vehicle for experimental development of interface systems. The results will be applicable to other mathematical programming procedures (e.g. non-linear and dynamic programming), since their data requirements are quite similar.

I.3. Man-Machine Communication

The challenging characteristics of problems for which computers are appropriate imply some form of interaction between a problem environment and a computer environment.

In some cases this interaction is direct, i.e. the computer environment is linked directly to the problem environment. Examples include on-line process control and routine data processing, including applications in which mathematical programming algorithms are built in to allow the computer to make automatic choices among alternatives.

In other cases, the communication is not direct, but is accomplished by the interference of the human. In this case the human "processor," using knowledge and experience, identifies the problem and expresses it in a formal notational language (e.g. of mathematics or chemistry). Operations research applications often fall in this category. Figure 1 illustrates the total communication process.

It is possible to generalize and extend the above concepts by looking closer into the spectrum of communication levels between the two extremes: the problem environment and the computer environment. Figure 2 displays this spectrum.

Since the digital computer is based on binary operations, the lowest communication level is machine language--strings of binary digits. Higher levels of input have been developed based on other numerical systems (decimal, octal, etc.), which generally decrease the interaction effort. The next level is represented by assembly languages, in which abbreviations (macros) of commonly-used sequences of binary instructions are used, taking on the characteristics of a limited lexicon of words and sentences in a natural language. High-level languages are next: such languages as Fortran or Cobol allow programmers to express in a single instruction the specifications of a task consisting of tens or hundreds of binary instructions.

Furthermore, more sophisticated interfacing systems have been developed. The rules concocted by a programmer

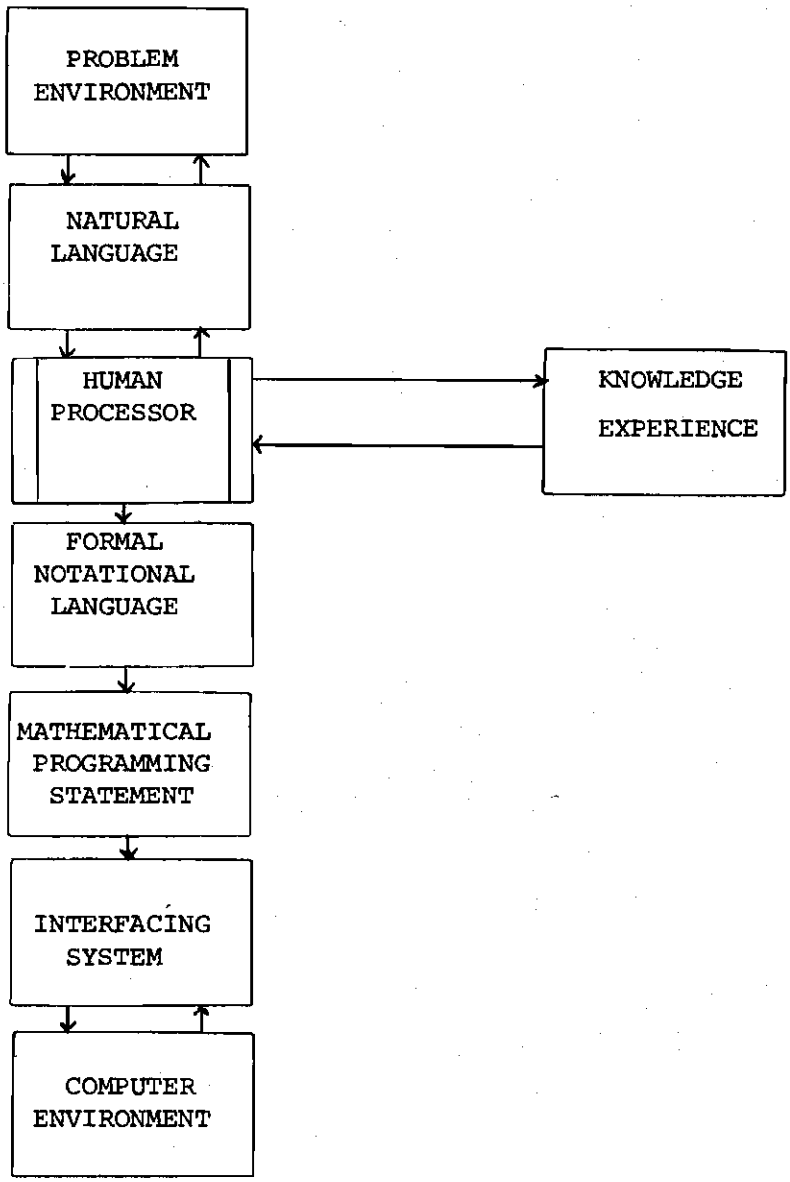


Figure 1. The Problem Formulation Process for Mathematical Programming

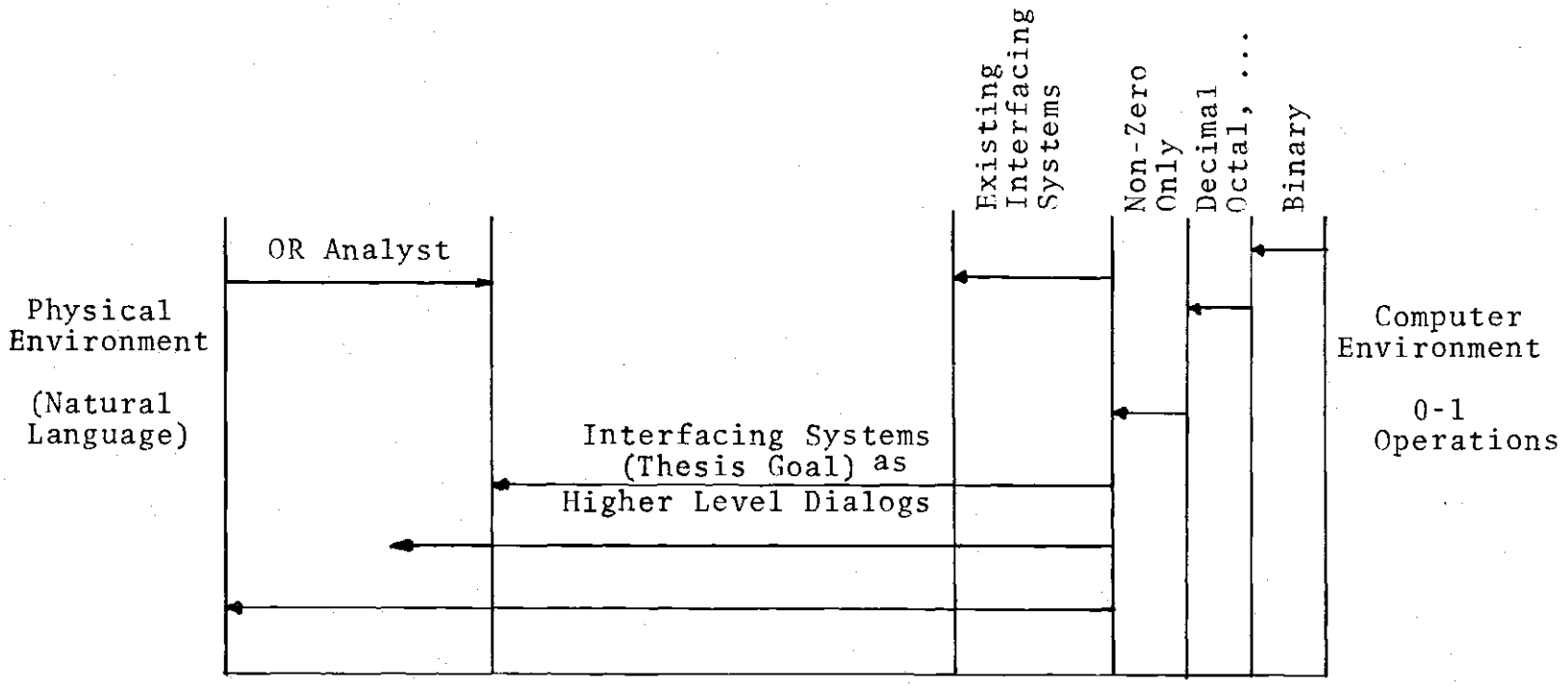


Figure 2. The Thesis Goal in the Spectrum of Interfacing Systems

for specifying data and instructions to a particular applications program can be considered as constituting an ad-hoc language. These systems provide the user with the facility of explicitly entering variables and mathematical expressions. Thus, a mathematical model can be entered in a real life form.

However, several limitations are present in the existing interfacing systems. An example of an existing interfacing system is the Multi-Purpose Optimization System (MPOS) developed at Northwestern University.

The analyst interacts with the environment and formulates the problem. Then he expresses it in a machine readable form and inputs it in an existing interfacing system.

The goal of this thesis consists of narrowing the gap between the environment and the machine. The gap being treated is the gap between the physical language at the problem environment and the computer readable language, which is the formation closest to the computation model. In more general terms the interaction is a triangle: (problem-human-machine). This thesis does not treat the gap between the human and the problem. We assume that the human formulates the problem in problem physical language. The research will concentrate on cost-effective interfacing systems; however, concepts of higher level dialogs which penetrate the area of problem formulation and the problem environment will also be explored.

Actually this gap in the spectrum represents several

things -- the lack of user-oriented features such as input flexibility, algorithm accessibility, i.e. capability for a user to access several algorithms with a simple input procedure, requirement for computer skills on the part of the user, and various economical aspects of the man-machine communication.

I.4. Outline of the Thesis

Chapter II presents the survey of several systems for linear and quadratic programming. The presentation concentrates on the interfacing part of the system.

Chapter III presents fundamental concepts and definitions from the theory of formal grammars, languages and automata. This chapter provides a general background related to the topic of the thesis. It is not a complete presentation of the above mentioned theories.

In Chapter IV we discuss the design and implementation of the interfacing system. The discussion is concentrated on the methodology and the application of the concepts presented in Chapter III.

In Chapter V we present the economic analysis of interfacing systems. Cost and benefit considerations are defined and discussed. A cost-benefit model for system evaluation is presented.

Chapter VI discusses the user psychology. Psychological considerations of man-machine communication are presented

and their impact on the effectiveness of the system is discussed. Certain characteristics of the human short term memory are also discussed.

Chapter VII presents the application of the methodology and concepts presented in the previous chapters. The discussion is initiated with linear programming and a particular system (EZLP) is presented. The applicability of the concepts is illustrated through non-linear programming.

Chapter VIII presents the conclusions of this thesis and discusses topics for further research.

CHAPTER II

REVIEW OF EXISTING INTERFACING SYSTEMS FOR LINEAR PROGRAMMING

This chapter presents a discussion of available systems for mathematical programming. The presentation will concentrate on interfacing and input. The features and limitations of each system will be identified and discussed. The discussion will concentrate on features affecting the rate of communication, information capacity of the input and ease of learning the particular language.

The spectrum of computer systems for mathematical programming begins with the commercial production packages and ends with the high-level input systems. The former represents the closest point to machine level communication. The latter is the closest point to the problem environment.

The tracing of the development of man-machine communication, feature wise rather than time wise, should begin with a typical commercial package.

A typical production package for mathematical programming is given by UNIVAC 1108-LP.

This system has extensive capabilities which assume a considerable familiarization time for the user. The UNIVAC 1108-LP is supported by a rather complex control

language with structure similar to that of a general purpose language. The commands of the system provide an extensive variety of combinations.

The input of the problem is quite lengthy since it is required to be inputted as one numeric per card, i.e. a 10 × 15 problem will require 150 cards only for the data deck. Even the zeros must be specified. This system is more intended for system-to-system communication with the data generated by a program.

If the 1108-LP were used, in an interactive mode, it would provide an extremely low rate of communication. This would severely damage the cost effectiveness and the overall utility of the system.

A class of systems next in user oriented features improvements to the commercial packages is well represented by LINDRU*** developed at Dartmouth College with support from General Electric.

The system has the following characteristics:

1. It does only maximization,
2. The user enters only the numeric entries of the model (row and column identifiers or variable names are not necessary),
3. The zeros must be specified,
4. The size of the problem and the number per kind of constraint must be specified in advance,
5. The constraints must be in a specific sequence:

upper bound inequalities, equalities, and lower-bound inequalities.

This system represents a considerable improvement in the area of man-machine communication for linear programming. Certainly it is more efficient than the production systems, but it imposes serious restrictions on the sequence and format of the input. Also, the system possesses a low degree of information compression, affecting the rate of communication.

Conversational Linear Programming (CLP) [51] was developed at the University of Texas at Austin. The system is a user oriented LP package. As such it allows easy changing and entrance of the problem. Certain key words are used to describe the kind of input. The following key words are available: CLEAR, TITLE, LABELS, MAXIMIZE, MINIMIZE, CONSTRAINT, BOUND, PRINT MATRIX, SOLVE, PRINT SENSITIVITY, PRINT DUAL, CHANGE, STOP. An asterisk must appear before these key words.

The default variable names are X1,X2,...,XN. If it is desired to use different names than these the user can specify them in advance via the command LABELS. The model is entered as a sequence of numbers, including the zeros, separated by blanks. Only one objective function is allowed and the key word MAXIMIZE or MINIMIZE must be entered before it. The key word CONSTRAINTS must be entered before the constraint. Constraint names are allowed. The default name is ROW.

The following is a sample model:

```
*LABELS AWCT BWCT ARJ ATJ ATP BTP
*MAXIMIZE
*82.2 82.2 66.0 66.0 74.0 74.0
*CONSTRAINTS
*A-TOM:  1 0 1 0 1 0 LE 600
*B-TOM:      0 1 0 1 0 1 LE 2400
*TJ-MAR: 0 0 1 1 0 0 LE 1000
*TP-MAR:   0 0 0 0 1 1 LE 2000
*WCT-QU: 0 0 3 -1 GE 0
*TJ-QU:0 0 3 -1 GE 0
*SOLVE
```

Several printing facilities are available and the system produces a comprehensive output with user control over the extensiveness of output reports.

Editing facilities are also available and the user can directly change entries in a line without re-entering the line. For instance:

```
*CHANGE
*A-TOM RHS 700
*SOLVE
```

Upper bounds can be assigned to the variables via the key word BOUND followed by variable name-value pairs.

An early system which allows mathematical expressions is the LP package developed in the University of Illinois, Urbana [50] in 1964. This system can be considered as the

first significant step, beyond the production packages, with respect to the input. The vocabulary is limited to two key words: MAXIMIZE and MINIMIZE. Arithmetic expressions are entered in fixed instead of free format. However, the system is very simple and requires noninstruction. The following is a sample model.

```
LINEAR PROGRAM FOR (1) PROBLEM
MAXIMIZE THE FUNCTION = .5X(1) + 6X(2) + 5X(3),
CONSTRAINT (1) 4X(1) + 6X(2) + 3X(3) LE 24,
CONSTRAINT (2) X(1) + 1.5X(2) + 3X(3) LE 12,
CONSTRAINT (3) 3X(1) + X(2) LE 24,
END OF PROBLEM
```

UHELP (University of Houston Easy Linear Programming) was developed in the University of Houston in 1969 [16]. UHELP represents an interesting trade-off between power and simplicity of learning. The system provides a higher degree of flexibility than the systems presented until now. Specifically, it allows:

i. meaningful variable names

ii. up to five objective functions in the same model

UHELP provides editing facilities through the key words CHANGE, DELETE, ADD.

The syntax of these commands is:

```
CHANGE:  <constraint name> : <new constraint>
CHANGE:  <obj. fcn. name>  : <new obj. fcn.>
CHANGE:  <obj. fcn. name>  : <variable name> : <new coef.>
```

CHANGE: <constraint name> : <variable name> :
<new coefficient>

ADD: <name> : <constraint>

DELETE: <name>

The key words MAXIMIZE and MINIMIZE require as argument an objective function name. This implies that the optimization can be directed to different objective functions without reentering the mathematical expression.

The following is a typical formulation in UHELP:

LATHE: 3.5 P1 + 4.6 P2 < 90

GRINDER: 4.3 P1 + P2 + 2.8 P3 < 50

DRILL: 7.5 P1 + 2P 2 + 3.0 P3 < 160;

SALES: P2 < 25;

OBJECTIVE: PROFIT: 17.00 P1 + 6.9 P2 + 7.5 P3;

MAXIMIZE : PROFIT; STOP;

Although the system presents an improvement "to" or "on" the previously presented systems, it still lacks input features to substantially affect the rate of communication. For instance, consider the situation that several variables are to be assigned the same upper or lower bound. In UHELP this would be accomplished by a list of separate constraints.

A broader system is MPOS (Multi Purpose Optimization System) developed at Northwestern University [13]. MPOS provides facilities for linear integer and quadratic programming. In addition the system can create interfacing data banks for the Control Data's large scale mathematical

programming system (APEX I, II).

A problem can be entered in MPOS either in its algebraic form or in a matrix form. This facility enables MPOS to be linked to model generating software. The system is supported by a rich vocabulary of key words.

With respect to the constraints, MPOS provides the user with the same degree of flexibility as UHELP. However, in MPOS the variable names should be specified in advance. Bounds must be entered explicitly, i.e., one variable per constraint. The following two sample problems illustrate the power and deficiencies of MPOS.

```
Column 1
VARIABLES
X1 TO X3
MAXIMIZE
12X1 + 5X2 + 7X3
*     OBJECTIVE FUNCTION IN $ (PROFIT)
CONSTRAINTS
5X1 + X2 + X3   .LE.   80
2X1 + X2 + 2X3     .LE.   100
BOUNDS
X3   .LE.   15
RNGRHS
OPTIMIZE
STOP
```

```
Column
1
WOLFE
VARIABLES
X1 X2
MAXIMIZE
X1 + X2 - 0.5X1 * X1 + X1*X2 - X2*X2
CONSTRAINTS
X1 + X2     .LE.     3
2X1 + 3X2          .GE.     6
PRINT  1
OPTIMIZE
STOP
```

A series of interactive linear programming programs was developed in the School of Textile Engineering at Georgia Institute of Technology. This series represented an effort to increase the conversational power of an existing linear programming program for academic support. The development was part of a thesis research by Coff [14] and of a project sponsored by Camsco [11].

The final product of the series of programs is called LP6 and provides the following features:

1. The model is inputted as a series of numerics.
2. The user can indicate the type of constraints by +1, 0, -1 for $\leq$, =, $\geq$, respectively.
3. Provides the following editing commands for

alteration of objective function coefficients, constraints and matrix elements: ALT COF, ALT RS, ALT MC.

4. Provides the following editing commands for adding or removing variables or constraints: ADD VAR, REM VAR, ADD CONSTR, REM CONSTR.
5. Provides the following control commands: PROBLEM, DISPLAY, SCORE, SOLVE, SOLVE INT, END.
6. The program can solve integer linear programming problems.

A version of LP6 has the capability to read the problem formulation from data files. These data files were created by another program responsible for retrieving the problem specifications and formulating the model.

In Chapter VII, EZLP will be presented. EZLP relaxes most of the restrictions of the above systems and provides the user with features as free format and high data compression as well as a variety of simplex based procedures. Before discussing EZLP we will discuss a language in general, the associated user psychology and the economics of interfacing systems design.

CHAPTER III

FUNDAMENTALS OF FORMAL GRAMMARS, LANGUAGES AND AUTOMATA

III.1. Introduction

This chapter presents a framework for the design of cost-effective interfacing systems. Since the core of an interfacing system is its language, elements of the theory of formal grammars and languages will be presented. A grammar is a mathematical model that generates the language of interest in the sense that the grammar specifies which formal manipulations of elements of the language are allowable. The grammar, as such, offers the foundations for an analytical study of the corresponding language. This is of primary importance in language design as well as for the design of the corresponding parsing algorithm. A parsing algorithm examines a sequence of symbols and reexpresses it in a given language, or determines that the sequence violates the grammar. In this thesis, "parsing" will refer to examining human-generated sequences of keystrokes and converting them to expressions in machine-readable language.

With respect to mathematical models and programming languages in general, certain types of grammars will be discussed and their potential for precise description of a language will be illustrated.

III.2. Concepts and Definitions

III.2.1. Formal Systems

A formal system, often called axiomatic system, is defined as a logistic system. It consists of an alphabet, a set of axioms and a finite set of relations.

Formal systems provide us with a powerful means of modeling intuitive notions which result in a rigorous analytical study of the behavior, properties, and potential applications of these notions. One of these applications is the automatic processing of languages. Formal systems are important for the language and particularly for the syntax specifications.

To process a language by a feasible method, a clear description of the language is needed. Hence, a language is needed to describe that language. The latter is called an object language and the former a meta-language. A formal system is a meta-language.

The symbols of the object language are called terminal symbols and the symbols of the meta-language are called non-terminal symbols. In this thesis the non-terminals will be indicated by capital letters and the terminals by small letters.

Donovan [17] provides the following definition:

DEFINITION 1. An alphabet A is a finite set of terminal symbols.

The following notation is used for alphabets.

$A^0 = A$

$A^1 = A$

$A^{n+1} = A^n \cdot A^1$

A^* designates all possible finite strings on an alphabet A. A^* is called the closure of A. Hence, $A^* = \bigcup A^n$, $n \geq 0$

EXAMPLE 1. Let $A = \{a\}$

then $A^* = \{a, aa, aaa, \ldots\}$

DEFINITION 2. A formal system is an ordered triple (A,S,P) where

1. A is the alphabet of the system
2. S is a set of words (Strings on A) called axioms.
3. P is a set of n-place relations on A^* with $n \geq 2$ (i.e. at least pairs).

DEFINITION 3. A production is a string transformation rule. It has a left-hand side which is the string to be transformed and a right-hand side which is the transformed string. To have a production, at least a substring of positive length of the left-hand side must be transformed.

DEFINITION 4. A starting symbol is a unique non-terminal from which all the string of a language are derived. (A terminal is a symbol that may be thought of as incapable of simplification.)

III.2.2. Formal Grammars and Languages

A language, natural or artificial, can be described as an infinite set of strings on some finite alphabet A. This set of strings is a proper subset of the closure A^* of the alphabet. Since there is an infinite number of possible strings we should look towards a specification of the patterns of the strings, i.e., the generic forms that generate the strings, assuming that there is a finite number of patterns. These patterns are the production rules as defined in Definition 3, and they are the core of a grammar.

A grammar is a model of a formal system as defined in Definition 1. In fact a grammar is an extended formal system. The closed form of a grammar definition is accomplished by the recursive description of the syntactical constructs or non-terminals. Non-terminals are syntactically structured in the sense that they can be broken down to other class structured non-terminals and/or eventually to terminal symbols, i.e. to symbols with no syntactical structure.

In a formal description of a grammar for a mathematical model, the non-terminal symbol CONSTRAINT may be used. This non-terminal might be defined in terms of other less structured non-terminals such as:

ARITH. EXPRESSION, REL. OPERATOR, NUMERIC

Breaking down these non-terminals we can define:

1. ARITH. EXPRESSION by VARIABLES and ARITHM. OPERATOR.
2. REL. OPERATOR by terminals > =, =, < =.

3. NUMERIC by SIGNED INTEGER, INTEGER.

Eventually, VARIABLES will be defined in terms of alpha-numeric characters, INTEGER by digits, etc. For simplicity, in some examples, single capital letters will be used to designate non-terminals and single small letters to designate terminals. For instance, CONSTRAINT could be represented by C.

Gross et al. [24] provide the following definition:

DEFINITION 5. A Grammar G is an ordered 4-tuple

$G = (N, A, \Sigma, P)$

where

1. N is the set of non-terminals
2. A is the set of terminals (alphabet)
3. Σ is the starting symbol with $\Sigma \varepsilon N$
4. P is the set of productions with $P \subseteq (NUA)^{*2}$
5. N A is empty

As defined above, P is the set of 2-place relations <a,b> with a, b ε $(N \cup A)^*$, where the symbol ε means "belongs to." The relation <a,b> is usually written as $a \rightarrow b$

EXAMPLE 2. Let G = (N,A, EXPRESSION, P)

with N = {FORMULA,B,C}, A = {b,c}

P = {EXPRESSION $\rightarrow$ BC, B $\rightarrow$ bB,

B $\rightarrow$ b, C $\rightarrow$ cC, C $\rightarrow$ c}

The language produced by the above grammar consists of strings of the term: $(b)^* \cdot (c)^*$. Thus, the string bbcc

belongs to the language as it is clear from the following sequence of productions:

bbbcc → bbbcC → bbbC → bbBC → bBC → BC → EXPRESSION

In operations research (OR) languages, the starting non-terminal Σ, which is the top of the syntactical tree, will be the term MODEL. This is the most structured non-terminal. Thus, we write G = (N,A,MODEL,P). In programming languages, Σ is the PROGRAM.

EXAMPLE 3. The following grammar is the basic generic form of each OR model. In a later section, a more detailed description of this grammar will be given. The symbol — is used only to separate grammatical symbols and it is not part of the grammar.

G = (N,A,MODEL,P)

A = {Alphanumeric characters, *,+,-,
>=, =, <=,ST,MIN,MAX}

N = {MODEL,OBJ. FCN,CONSTR. SET,
OPTIMIZE, ARITHM. EXPR., REL. OPER.,
NUMERIC, VARIABLE, CONSTRAINT,
LETTER, ALPHANUM}

P = {MODEL → OPTIMIZE __OBJ. FCN
__ST__CONSTR. SET
OBJ. FCN → ARITHM. EXPR.
CONSTR. SET → CONSTRAINT

CONSTR. SET → CONSTRAINT __CONSTR. SET

CONSTRAINT → ARITHM. EXPR. __REL.

OPER. __NUMERIC

ARITHM. EXPR. → VARIABLE

ARITH. EXPR. → VARIALBE __ARITH.

OPER. __ARITH. EXPR.

OPTIMIZE → MIN

OPTIMIZE → MAX

REL. OPER → > =

REL. OPER. → =

REL. OPER. → < =

ARITHM. OPER → +

ARITH. OPER → -

VARIABLE → LETTER

VARIABLE → LETTER __ALFANUM

ALFANUM → DIGIT

ALFANUM → LETTER

ALFANUM → DIGIT __ALFANUM

ALFANUM → LETTER __ALFANUM

NUMERIC → number

NUMERIC → number __NUMERIC}

Wall [52] provides the following definitions:

DEFINITION 6. If a string x in a grammar G is immediately derived from a string y, y = x is written.

DEFINITION 7. If a string X in a grammar G is derived from a string y through a sequence of derivations, $y \Rightarrow^* X$ is written, i.e. $y \Rightarrow z_o \Rightarrow z_1 \Rightarrow \ldots \Rightarrow z_{n-1} \Rightarrow z_n = x$

DEFINITION 8. A sentential form in a grammar G is any string which can be derived from the starting symbol Σ. In the Example 2 $\Sigma \Rightarrow BC \Rightarrow bC$, where bC is a sentential form.

DEFINITION 9. A sentence is a sentential form in G containing only terminal symbols. $\Sigma \Rightarrow BC \Rightarrow bC \Rightarrow bc$, where bc is a sentence.

DEFINITION 10. A language L defined by a grammar G (written L(G)) is the set of sentences derived from Σ in G. Hence, $L(G) = \{x \in A^* \mid \Sigma \Rightarrow^* X\}$

As defined above, a language L(G) is a set of well defined constructions and meaning.

DEFINITION 11. The syntax of a language L(G) is the set of rules specifying legal constructions of the language.

DEFINITION 12. The semantics of a language L(G) is the assignment of meaning to the constructions of the language.

For example, the position of the words MIN or ST is determined

by the syntax of the language, while the effect of these key words is determined by the semantics of it. The meaning of a construction in a given language is a correspondence of that construction to a construction in a different language, the different language assumed known.

III.2.3. Automata

In the preceding pages, definitions and elements from the theory of formal systems and particularly from the theory of formal grammars have been stated. This theory is used for automatic language processing, and in this thesis it will be applied in the processing of languages related to operations research such as mathematical models.

A theory closely related to the theory of formal grammars is the Theory of Automata. Automata are also used for language processing.

An automaton is an abstract device. The analogous mechanical device of the principal automaton model consists of an input tape, a read head (input scanner) and a control unit. Each automaton is characterized by a finite set of states. One of these states is an initial state and a proper subset of the state set consists of the terminal states. The device receives an input. When given a set of instructions, which are programmed in the control unit, it performs on the input a sequence of elementary operations in a stepwise fashion.

These steps of operation, which are the states of the

automaton, establish the recursive function of the automaton. The output of the current step is the input to the next step.

An automaton is designed to accept (let through) a specified class of input strings. The control unit of the automaton which is programmed with a set of instructions performs the proper state transition depending on the current symbol of the string under scanning. The set of instructions reflects the syntactical rules of the language.

An automaton, depending on the grammatical complexity of the language it recognizes, may have one or two-way input scanners (forward, forward-backward) and peripheral memory units.

The above structure of the automaton is supported by the following operational rules:

1. An item (substring of unit length) of the input is accepted if the control unit can perform a state transition on this item.
2. A substring of the input string is accepted if the automaton reaches a terminal state. The remaining part of the input string (possibly of zero length) is ignored. This may be employed as a comment field.
3. An input string is rejected if either
   i. the scanner reads an item with no corresponding state transition instruction (illegal syntax),

or

ii. the scanner reads the entire string without the automaton reaching a terminal state (incomplete input).

In a later section, automata theory will be looked at more closely.

The equivalency (association) of certain automata to certain classes of languages will be presented. A class of languages is equivalent to a specific type of automaton if this automaton recognizes languages of this class.

Automata and their relationship to formal grammars and languages provide a rigorous practical and theoretical framework for the development and/or evaluation of language recognizing algorithms.

The practical aspect of automata theory here is that a language can be associated to an automaton with specified sets of states and state transition instructions. After this step is accomplished, the operation of the model can be simulated by a computer code. That is, a parsing algorithm can be designed.

The theoretical aspect provides the means for studying the cost-effectiveness of the parsing process, and, thereby of the language. Modifications of the grammar of a language might cause variations of the overall efficiency of the system.

III.2.4. Syntax Specification

In Section III.2.1 we discussed the need for a meta-language to describe a language. In the following sections, the concepts of formal grammars, languages, and automata were introduced.

A more effective way of describing a grammar is needed. A meta-language for grammatical specifications is needed. A meta-language widely used to specify suntax is the Backus-Naur Form (BNF). It precisely specifies syntactical rules, but lacks the power to specify semantical rules.

In BNF the following notational rules exist:

1. Non-terminals are written in brackets (<>)
   i.e. <X> means the class of non-terminals named X.
2. The sign of production (→) is replaced by :: = and reads as "is replaced by."
3. Multiple ways of transforming a non-terminal (alternative productions) are written in the same line separated by the ORing operator "|".

EXAMPLES:

1. Very often in a grammar, the non-terminal DIGIT is needed. Possible productions are:

   DIGIT → 1

   DIGIT → 2

   .
   .
   .

   DIGIT → 0

In BNF, the following is written:

<DIGIT> :: = 1 | 2 | 3 ... | 9 | 0

In BNF, the recursive character of a set of productions can be easily specified, as in

<LIST> :: = <VARIABLE> | <LIST>, <VARIABLE>

An extension of the above feature is:

<LIST> :: = VARIABLE | <LIST>, <VARIABLE>
(n)

where (n) assigns an upper bound to the recursion.

The grammar in EXAMPLE 2, Section III.2.2 is written as follows:

<Σ> :: = <B> <C>

<B> :: = b | b <B>

<C> :: = c | c <C>

The following example could be a statement in a user oriented mathematical programming system. It illustrates the power of BNF and provides the foundation for some important observations. Consider: LET X1, X2, LINE3, POWER > = 10.5.

In the context of formal languages, this is a phrase of a language with a specified grammar. Part of this grammar is the following segment of syntactical rules related to the above phrase.

<LIST CONSTRAINT> :: = LET <NULL STRING><LIST><REL. OP><NUM>

<LIST> :: = <VARIABLE> | <LIST>, <VARIABLE>

<VARIABLE> :: = <LETTER> | <LETTER><LITERAL>
(k-1)

<LITERAL> :: = <DIGIT>|<DIGIT><LITERAL>|<LETTER>|<LETTER><LITERAL>
(k-2) (k-2)

<NULL STRING> :: = SPACE | SPACE <NULL STRING>
(4)

<NUM> :: = <INTEGER> <INTEGER>.<INTEGER>

<INTEGER> :: = <DIGIT>|<DIGIT><INTEGER>

<LETTER> :: = A | B | ... | Z

<DIGIT> :: = 1 | 2 | ... 9 | 0

It is observed that this grammar specifies a maximal length of K characters for variables. Also, it allows up to five spaces after the key word LET.

In BNF, user oriented features can be included or excluded by a quick transformation of the grammar. Hence, the language designer can visualize the effect of certain features. The effect has two components:

1. The user-response and attitude towards certain syntactical rules
2. The cost associated with the design and utilization of a parsing algorithm

As a final example of this section, a complete mathematical model and its grammar will be presented. Consider the following linear model:

$$\text{OPT} \quad \sum_{I=1}^{N} C_I X_I$$

$$\text{ST} \quad \sum_{I=1}^{N} A_{IJ} X_I = R_J \qquad J = 1,M$$

The corresponding grammar is:

G = (N,A,MODEL,P)

A = {A,B,...,Z,1,2,...,9,=,-,ST,=,MIN,MAX,NULL}

N = {MODEL, CONST. SET, CONST, OPT. AE, NUM, SIGN,
ARITHM. OP, VAR, AE2, LETTER, DIGIT, INTEGER,
LITERAL}

where AE denotes Arithmetic Expression and CONST denotes CONSTRAINT

P is the set of productions.

The grammar of the model is shown below in BNF

<MODEL> :: = <OPT><AE>ST<CONST. SET>

<CONST. SET> :: = <CONST.>|<CONST.><CONST. SET>
(M-1)

<CONST> :: = <AE> = NUM

<AE> :: = <SIGN><NUM><VAR>|<SIGN><NUM><VAR><AE2>
(N-1)

<AE2> :: = <ARITHM OP><NUM><VAR>|<ARITH. OP><NUM><VAR><AE2>
(N-2)

<VAR> :: = <LETTER>|<LETTER><ALFANUM>

<ALFANUM> :: = <DIGIT>|<DIGIT><ALFANUM>|<LETTER>|<LETTER><ALFANUM>
(k-2)

<NUM> :: = <INTEGER>|<INTEGER>.<INTEGER>

<INTEGER> :: = <DIGIT>|<DIGIT><INTEGER>

<DIGIT> :: = 1 | 2 | ... 9 | 0

<LETTER> :: = A | B | ... | Z

<OPT> :: = MAX | MIN

<SIGN> :: = + | - | null

<ARITHM. OP> :: = + | -

In this grammar

M is the number of constraints

N is the number of variables

K is the maximum number of characters in a variable

III.3. Elements from the Theory of Formal Grammars and Languages

III.3.1. Types of Grammars

In the previous section, the concept and potential of grammar transformations were introduced. A grammar transformation is a generalization of the concept of production since a production is defined as a string transformation. This definition of production permits a variety of string transformations. Hence, grammar transformations permit a variety of grammars with different structure. However, this is an unorganized and possibly infinite set of grammars. In the space of grammars, the following will be considered:

1. Identification of the sub-spaces of grammars that generate the languages of interest
2. Examination of these grammars in an organized fashion.

Considering the classical approach for constraining a space, we restrict the rules (productions) that define the grammars. The restrictions must be such that all of the grammars of a specific linguistic interest are included. Having redefined the global space, we can apply the same

approach recursively to define new subspaces of more specific interest.

Noam Chomsky has defined the following proper key restrictions on the production rules that constraint the initial global space into three inclusive subspaces. Wall [52] indicates that according to the Chomsky system the most general grammar is the one with no restrictions on the productions. This is called a type 0 grammar.

The first restriction is to require the right-hand side of a production to have at least as many symbols (terminals or non-terminals) as the left-hand side. This is called a type 1 grammar or context sensitive grammar. The context sensitivity of the grammar is clear if a production is a tree, where each node represents a symbol. This demonstrates the interdependencies of the trees of a grammar. The production AB → aAB is of type 1.

The second restriction is to require the left-hand side of a production to have a single nonterminal symbol. This is called a type 2 grammar or a context-free grammar since there are no interdependencies of the subtrees in the syntactical tree.

The production A → aAB is of type 2.

The productions VARIABLE → LETTER __VARIABLE and

CONSTRAINT → ARITHM. EXPRESSION_>=_NUMERIC are of type 2.

The third restriction is to require the left-hand side of the production to have a single non-terminal and the

right-hand side to have at most one non-terminal. This is called a type 3 grammar or a linear grammar.

The production A → aB is of type 3.

The production INEQUALITY → VARIABLE < = 5 is of type 3.

The production ARITHM. OPERATOR → + is of type 3.

At each step, each restriction is an addition to the previous restrictions. Thus, the types of grammars are inclusive so that each type n+1 is type n, but not all type n grammars are type n+1. This inclusiveness is illustrated in Figure 3 where the exterior bounds of the type 0 are not well defined.

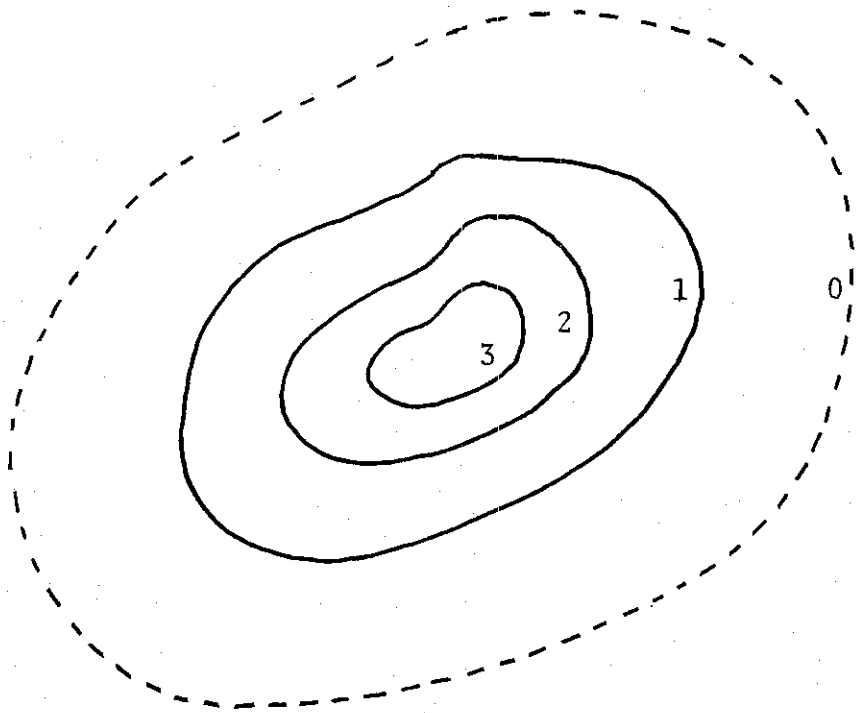


Figure 3. The Chomsky System for Grammar Classification

A grammar is specified as type n if n is the minimal type of production in the set of productions. The grammar of Example 2, Section III.2.4 will now be examined. Considering the left-hand sides of all productions, it can be seen that they have a single non-terminal. Hence, the grammar is at least type 2. Looking at the right-hand sides, it can be seen that there are several productions with more than one non-terminal. Hence, it is not of type 3. The conclusion is that this grammar is type 2.

In the scope of this thesis, the primary focus is on type 2 and 3 grammars. To make certain key observations related to types of grammars and to the associated parsing costs, the following assumptions will be made. It is assumed that the structure of a grammar directly affects the overall complexity of the associated parsing algorithm. The term overall complexity reflects the fact that complexity can be differentiated into space complexity and time complexity.

When introducing automata in Section III.2.3, it was noted that classes of languages are associated to specific forms of automata. These classes correspond to the types of grammars. The observation to be made here is a result of this framework:

i. In designing a grammar, the syntactic rules can be constructed so that the grammar will be of a specific type.

ii. In modifying a grammar, it can be shifted to a

different type to substantially affect the complexity of the parsing algorithm.

A second observation is that in the production set of a grammar, productions of different types are usually found. The production set is not uniform with respect to the type of productions. When considered with the fact that a parsing algorithm of different complexity corresponds to each type of production, we are naturally led to a decomposition of the grammar and design of multi-pass parsing algorithms.

It is not meaningless to discuss the degree of non-linearity of productions of type 2. The degree of non-linearity is the number of non-terminals in the right-hand side of the production. The grammar can be rearranged with a descending degree of non-linearity. This provides a picture of possible natural "cuts" of the grammar. Having rearranged the grammar, we can observe the "density of non-linearity." Accordingly, the analyzing system, a proper combination of recognizing automata, can be designed.

The inclusiveness of grammars applies to automata. Gross [24] indicates that automata recognizing type 2 languages can recognize linear languages. The inverse is not true. Linear analyzers can be described as automata recognizing linear productions and non-linear analyzers as the automata recognizing type 2 productions. It will be seen later that a non-linear analyzer has a higher time complexity and space complexity than the linear. This is

interpreted as a higher cost for a slower recognition.

The foregoing discussion is based on the observation that a particular Context Free (CF) grammar contained linear productions. By sorting the grammar by degree of non-linearity, it was decomposed to a CF and linear grammar. However, this was a particular case. The question of whether every CF grammar can be decomposed to a CF and a linear remains. The answer is not known. This is one of the unsolved problems in mathematical linguistics. The impact of the above, if it were true, would be very valuable since the density of non-linearity could be controlled by recursively transforming the CF to a CF' and a linear.

The problem is stated as follows:

$$CF_{d_k}^{n-1} \stackrel{?}{\rightarrow} CF_{d_e}^{n} + L^{n-1} \quad \text{where } d_k,\ d_e \qquad (1)$$

degrees of non-linearity. This in the limit, would imply the complete reduction of non-linear analysis. However, by this scheme, the density of non-linearity can be controlled, while the degree of non-linearity cannot be controlled, i.e., even if (1) holds it cannot be always said that $d_k > d_e$.

The density of non-linearity is related to the expected utility of the non-linear analyzer while the degree of non-linearity is related to operational cost of the non-linear analyzer. However, this approach cannot be formalized since (1) has not been proven. This approach, though can be

empirically justified from the fact that in some practical applications natural decomposition of CF grammars can be observed. In the environment of programming languages as well as of languages for operations research models, a set of linear productions can almost always be identified.

III.3.2. From Grammars to Languages

Every type n grammar generates a type n language which is unique for this grammar. However, every type n language may be generated by more than one different grammar. This is important for the simplification of grammars, because a grammar might be simplified and still generate the same language. A discussion on grammar simplifications is given by Wall [52].

III.4. Elements from the Theory of Automata

In Section III.2.3 the concept and structure of automata were introduced in a non-formal way. In this section the formal definitions of certain automata as well as their operation with respect to the language recognition will be presented.

It was indicated that the (4) four types of languages (Chomsky system) are associated to four kinds of automata. This association is shown below:

Type of Language	Recognizing Automaton
0	TURING MACHINES (TM)
1	LINEAR BOUNDED (LBA)
2	PUSH-DOWN (PDA)
3	FINITE (FA)

This association is supported by four theorems in Machematical Linguistics. The proofs of these theorems can be found in Aho and Ullman [2]. The language describing operations research models are of type 2 or 3. Hence, PDA's and FA's are the main interest of this thesis.

III.4.1. Finite Automata

The first three types of automata represent extended forms of the FA which is an automaton with the least power. This reflects the fact that each n-1 language has less restrictions in the production rules than a type n language and hence requires a more elaborate recognition algorithm.

The FA has a one-way (left to right) read-head and a control unit. It does not have any memory. The movement the FA makes is determined by:

i. the current state

ii. the current symbol under scanning

Given an input string to be recognized, it is assumed that there are no blanks in the string. This assumption is easily justified if a pre-scanner is considered. The pre-scanner scans the string and supresses the blanks in a way

that the positional relationships of the items of the string are not disturbed. This is one of the functions of a linear analyzer. Also, it is assumed that there are blanks at each side of the input string. These two assumptions will hold for the rest of this thesis.

As was informally described in Section III.2.3, the operation of an automaton is directed by its control unit which is properly programmed. This program is part of the definition of an automaton and consists of a set of instructions with the following format:

$$(a_i, q_j) \rightarrow q_k$$

where

i. a_i is an input symbol under the scanner

ii. q_j is the current state

iii. q_k is the next state

The instruction causes the following action by the control unit. If the automaton is at state q_j and the input symbol is a_i, the FA is moved to state q_k and the scanner is moved one position to the right.

According to the operational rule 1 of Section III.2.3, the symbol is accepted. The operation goes on in this fashion until the string is accepted or rejected according to the operational rules.

According to the definition of linear grammars, a legal production could be $A \rightarrow xBy$ where A, B are non-terminals and x,y are terminals. The regularity condition is satisfied

if x or y is an empty string. A way to satisfy this condition is, when constructing the linear rules of a grammar, to use left or right recursion and never to mix them in the same grammar.

It is convenient to represent a FA by a state graph where each node is a state at each arc. $\langle q_i, q_k \rangle$ is associated with a terminal symbol. We associate the nodes with the non-terminals and the arcs with the terminals, so that the production A → xC is translated to the instruction (x,A) → C. A "dummy state" or "final state," F, is used for the productions B → x. The corresponding instruction is (x,B) → F. F is not a linguistic element as the other states are. It is a state associated with all the terminal states and it is needed for the formal programming of the control unit. In practice, the state is bypassed and the operation is linked to the initial state, for the next string.

EXAMPLE 4:

Let G = (N,A,MODEL,P) where MODEL is the initial state

N = {MODEL,B,C,D}, A = {a,b,c,d}

P = {MODEL → bB, B → bc, c → aC, C → cD,

D → bD, D → d, B → a}

The symbol number (#) denotes the blank character. Figure 4 represents the graph of this grammar.

The set of instructions for this graph are:

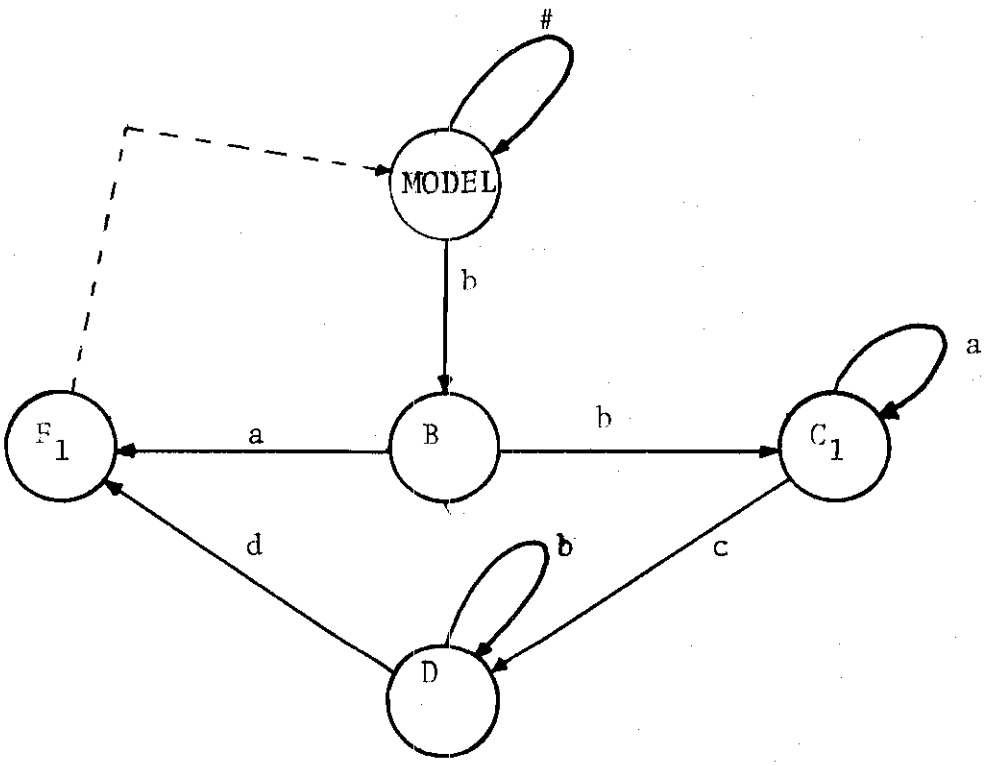
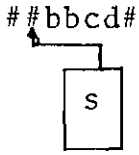
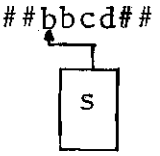
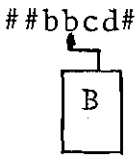
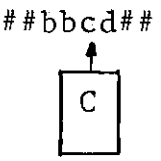
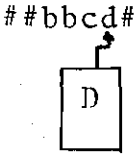
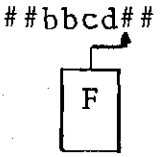


Figure 4. The State Graph of the Grammar of Example 4

1. (#,MODEL) → MODEL
2. (b,MODEL) → B
3. (b,B) → C
4. (a,C) → C
5. (c,C) → D
6. (b,D) → D
7. (d,D) → F
8. (a,B) → F

The following phrase of the above language will be considered:

Position of Scanner & State	Instruction Number	Position of Scanner & State	Instruction Number
	1 →		2 →
	3 →		5 →
	7 →		

Hence, according to the operational rules the string is accepted as a legal construct of this language. The above state transitions correspond to the following path in the syntactical tree.

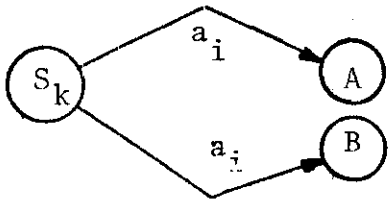
MODEL → bB → bbC → bbcD → bbcd

Consider the following string:

"bbcbd"

Position of Scanner & State	Instruction Number	Position of Scanner & State	Instruction Number
##bbcbd## MODEL	1 →	##bbcbd## MODEL	2 →
##bbcbd## B	3 →	##bbcbd## C	5 →
##bbcbd## D	? →		

At this point it is seen that there is not an instruction of the form (b,D) and hence the string "bbcbd" is rejected. An FA is called non-deterministic if one or more instructions (a_i,S_k) map to more than one state, i.e., $(a_i,S_k) \rightarrow$

{A, B, ...} or 

Gross et al. [24] provides the following definition:

DEFINITION: D-FA Deterministic Finite Automaton.

A Deterministic FA is a 5-tuple (K,A,M,S,T) where

1. K is the finite nonempty set of states
2. A is the input alphabet of the D-FA

3. M is a mapping function from A^* × k into K
   i.e. if M (w,q) = R then when in state q, if the D-FA receives as input the character (or string) w, it switches to state R
4. S is the start state with S ε K
5. T is a nonempty set of terminal states T ε K

A non-deterministic FA, (ND-FA), is defined as a, D-FA with a difference on item 3, which is:

3. M is a mapping function from A^* × K into subsets of K.

A method for converting ND-FA's into D-FA is given in Aho and Ullman [2]. To summarize this section the following statements are made:

STATEMENT 1. Finite Automata recognize exactly the class of Linear Grammars (type 3)

STATEMENT 2. ND-FA's and D-FA's are equivalent with respect to the language they recognize.

III.4.2. Push-Down Automata (PDA)

The information presented in this section was taken from Gross et al. [24]. Next in recognizing power automaton is the PDA. It has two operating heads, a control unit and a stack memory unit as a working area. The first head is a read one which reads from left to right. The second is a read-write-erase one and it operates on the stack memory. The movement of the PDA is determined by:

i. the current state

ii. the current input symbol

iii. the symbol last stored in the stack memory (top-most element)

At each point the operating unit with the two heads, reads the input symbol and the top-most element of the stack.

STATEMENT: The PDA recognizes exactly the class of CF languages. The proof of the above statement may be found in Aho, et al. [2]. Gross et al. [24] provide the following definition:

DEFINITION: A PDA is a 7-tuple: $(K,\bar{A},\Gamma,M,q_o,\sigma,F)$ where:

1. K is a finite nonempty set of states
2. $\bar{A}$ is the input alphabet of the PDA
3. Γ is the alphabet used in the stack memory called the push-down alphabet
4. q_o is the initial state, $q_o \in k$
   In the context of this thesis q_o will be referred by MODEL.
5. σ is a particular start symbol for the stack memory
6. F is the set of final states $F \in K$
7. M is a mapping function from $K \times (A \bigcup \{e\}) \times \Gamma$ to finite subsets of $K \times \Gamma^*$.
   where e is a neutral word used for convenience. This neutral word is added to A, where A is the alphabet of the associated CF language.

In the above definition:

$$\bar{A} = A \cup \{e\}$$

$$\Gamma = A \cup N \cup \{e\} \cup \{\sigma\}$$

where N is the set of non-terminals of the associated CF language.

The function M as defined above, establishes the non-deterministic character of the PDA. If M were defined as a mapping to a single element of $K \times \Gamma^*$, then the PDA would be deterministic.

III.4.2.a. Operational Definitions for the PDA. In order to describe the operation of a PDA as well as the programming of its control unit, the following definitions are needed.

1. A physical situation of a PDA is the triple (a_i, S_k, Z_j) where
   a_i is an input symbol
   S_k is a state
   Z_j is the top-most symbol in the stack memory
2. A situation of a PDA is either a physical situation or a triple obtained from a physical situation by replacing a_i or Z_j or both by the neutral word e. This word acts as a cover on a symbol so that it cannot be seen by the PDA. To each physical situation there correspond four situations.
   $S_1 = (a_i, S_k, Z_j)$
   $S_2 = (e, S_k, Z_j)$

$S_3 = (a_i, S_k, e)$

$S_4 = (e, S_k, e)$

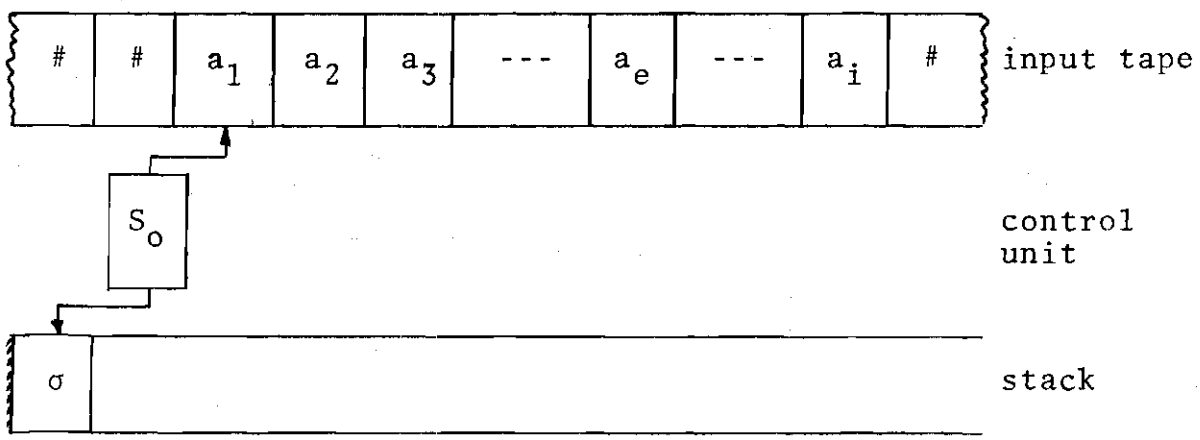
3. An instruction of the PDA has the following format: $s \rightarrow (S_m, x)$ where s is a situation, S_m is a state and $x \in \Gamma$

4. The degree of a string ϕ with $\phi \in \Gamma^*$ is an integer λ such that: $\lambda(\sigma) = -1$

$\lambda(e) = 0$

$\lambda(\phi) =$ # of symbols in ϕ

III.4.2.b. Operation of a PDA. At the beginning of the recognition process, the input string is on the input tape (buffer) with blanks at each side.



The PDA is in the initial state. One head is reaching a_1 and the top-most symbol of the stack is σ. Therefore, the operation begins with the physical situation (a_1, S_o, σ).

If at any physical situation s, there is no instruction, the automaton halts. If there are one or more instructions, the automaton chooses one of them.

Suppose, it is at $s \rightarrow (S_m, x)$. The PDA will perform the following moves:

1. The control unit passes state S_m.
2. If the first symbol of the situation is e, the first read head remains on the same input symbol.
3. If the first symbol of the situation is a_i, the read head reads the symbol to the right of a_i.
4. The second head operates ontthe stack memory as follows:

   (a) $x=\sigma = \lambda(\sigma) = -1$: Erase the top-most symbol in the stack and read the one under it.

   (b) $x=e = \lambda(e) = 0$: Do nothing

   (c) $x=\phi = \lambda(\phi) = K$: Enter the string x as the top-most symbol in the stack (push-down) and move the head K positions so that it reads the first symbol of x.

A string is accepted by the PDA if the operation terminates at the situation (#, S_o, #). A string is acceptable if there exists a computation that finishes at the situation (#, S_o, #). However, since the PDA is non-deterministic, we can have the situation that an acceptable string is rejected although another state-path might accept it. So, all the possible paths must be tried. With the finite automata

the deterministic or non-deterministic models were always recognizing the same language. This is not always true for the PDA's.

STATEMENT: Every CF language accepted by a non-deterministic PDA is not necessarily accepted by the corresponding deterministic PDA.

III.4.2.c. Constructing the PDA. The following generalized procedure constructs a PDA so that it accepts a DF language. It is given by Gross and Rentin [24].

Given the DF grammar G = (N,A,MODEL,P), let N_i be the non-terminals and q_i the terminals. The states of the PDA are indicated by brackets ([]) and they are the following:

i. an initial state [0]

ii. a pre-syntactic state [1]

iii. syntactic states [N_i]

The state transition instructions are of 5 types:

	Type	Format
1.	Initial instruction	(e, [0],) → ([1], MODEL)
2.	Instructions which switch to a syntactic state. As many as non-terminal symbols N_i	(e, [1],N_i) → ([N_i],e)
3.	Analysis instructions. As many as production rules in the grammar as $N_i \rightarrow x_i$	(e, [N_i],e) → ([1],x_i)

Type	Format
4. Read (or write) instructions as many as there are terminal symbols	$(q_j,[1],a_j) \rightarrow ([1],e))$
5. Terminal instruction	$(e,[1]e) \rightarrow ([0],e)$

EXAMPLE:

Suppose the following grammar is given:

G = ({MODEL, A, B}, {a,b}, {MODEL},P)

P = {MODEL → AB, A → B b, B → a}

The language generated by this CF grammar is recognized by a PDA with the following set of instructions:

1. $(e,[0],\sigma) \rightarrow ([1],MODEL)$
2. $(e,[1],MODEL) \rightarrow ([MODEL],\sigma)$
   $(e,[1],A) \rightarrow ([A],\sigma)$
   $(e,[1],B) \rightarrow ([B],\sigma)$
3. $(e,[MODEL],e) \rightarrow ([1],AB)$
   $(e,[A],e) \rightarrow ([1],Bb)$
   $(e,[B],e) \rightarrow ([1],a)$
4. $(a,[1],a) \rightarrow ([1],\sigma)$
   $(b,[1],b) \rightarrow ([1],\sigma)$
5. $(e,[1],\sigma) \rightarrow ([0],\sigma)$

EXAMPLE:

The following CF-grammar describes the basic structure of an inequality as it may be used in a mathematical model.

<INEQ> :: = <AE> <RO> num

<AE> :: = var | var <AO> <AE>

<RO> :: = >= | = | < =

<AO> :: = + | -

where

INEQ denotes Inequality

AE denotes Arithmetic Expression

RO denotes Relational Operator

AO denotes Arithmetic Operator

num denotes numeric

var denotes variable

In this example num and var are considered as terminal symbols.

The corresponding recognizing PDA will have the following instructions:

Type	Number	Instruction Set
1		(e,[0],σ) → ([1],INEQ)
2	2.1	(e,[1],INEQ) → ([INEQ],σ)
	2.2	(e,[1],AE) → ([AE],σ)
	2.3	(e,[1],AO) → ([AO],σ)
	2.4	(e,[1],RO) → ([RO],σ)
3	3.1	(e,[INEQ],e) → ([1],AE__RO__num)
	3.2	(e,[AE],e) → ([1],var)
	3.3	(e,[AE],e) → ([1],var__AO__AE)
	3.4	(e,[RO],e) → ([1],>=)
	3.5	(e,[RO],e) → ([1], =)

Type	Number	Instruction Set
	3.6	(e,[RO],e) → ([1],< =)
	3.7	(e,[AO],e) → ([1], +)
	3.8	(e,[AO],e) → ([1], -)
4	4.1	(num,[1],num) → ([1],σ)
	4.2	(var,[1],var) → ([1],σ)
	4.3	(+, [1], +) → ([1],σ)
	4.4	(-, [1], -) → ([1],σ)
	4.5	(>=,[1],>=) → ([1],σ)
	4.6	(<=,[1],<=) → ([1],σ)
	4.7	(=,[1],=) → ([1],σ)
5		(e,[1],σ) → ([0],σ)

III.4.2.d. The Formal State Graph of a PDA. As presented above, the mapping function of the PDA is always defined by a set of state transition instructions which are classified in five groups.

As in finite automata, each non-terminal symbol of the CF grammar will be seen as a state at the PDA. In addition, there is the initial state [0] and the presyntactic state [1].

The presyntactic state is a key state of the PDA. It is used as a "reference point" of the recognition process. The PDA switches to this state after executing:

i. the initial instruction (type 1)

ii. a production instruction (type 3)

iii. a read or write instruction (type 4)

By ordering the states on an axis the state graph of a PDA is drawn.

In Figure 5, A_i's are the non-terminals with A_1 = MODEL. a_j's are the terminals. The circles indicate states while the squares indicate the presyntactic state (reference point). The w_i's are the right hand-side of productions, i.e.

$$A_i \rightarrow w_{i,1} \qquad A_i \rightarrow w_{i,2} \qquad \ldots A_i \rightarrow W_{i,k}$$

the notation x|y|z is used to describe the corresponding operation, where

x is the current input symbol

y is the current top-most symbol in the stack

z is the new top-most in symbol in the stack

This triple corresponds to the instruction (x, [s], y) $\rightarrow$ ([s'], z) where s' can be s. It is easy to expand this model and construct the state graph of a PDA recognizing a given CF language. The expansion takes place in Sections II.3.4. For each additional non-terminal, a type 2, as many type 3 instructions (arcs) and as many transformations of the non-terminal are added. For each additional terminal, one instruction (arc) of type 4 is added.

III.5. Summary

The following set of statements summarizes the important points of this chapter:

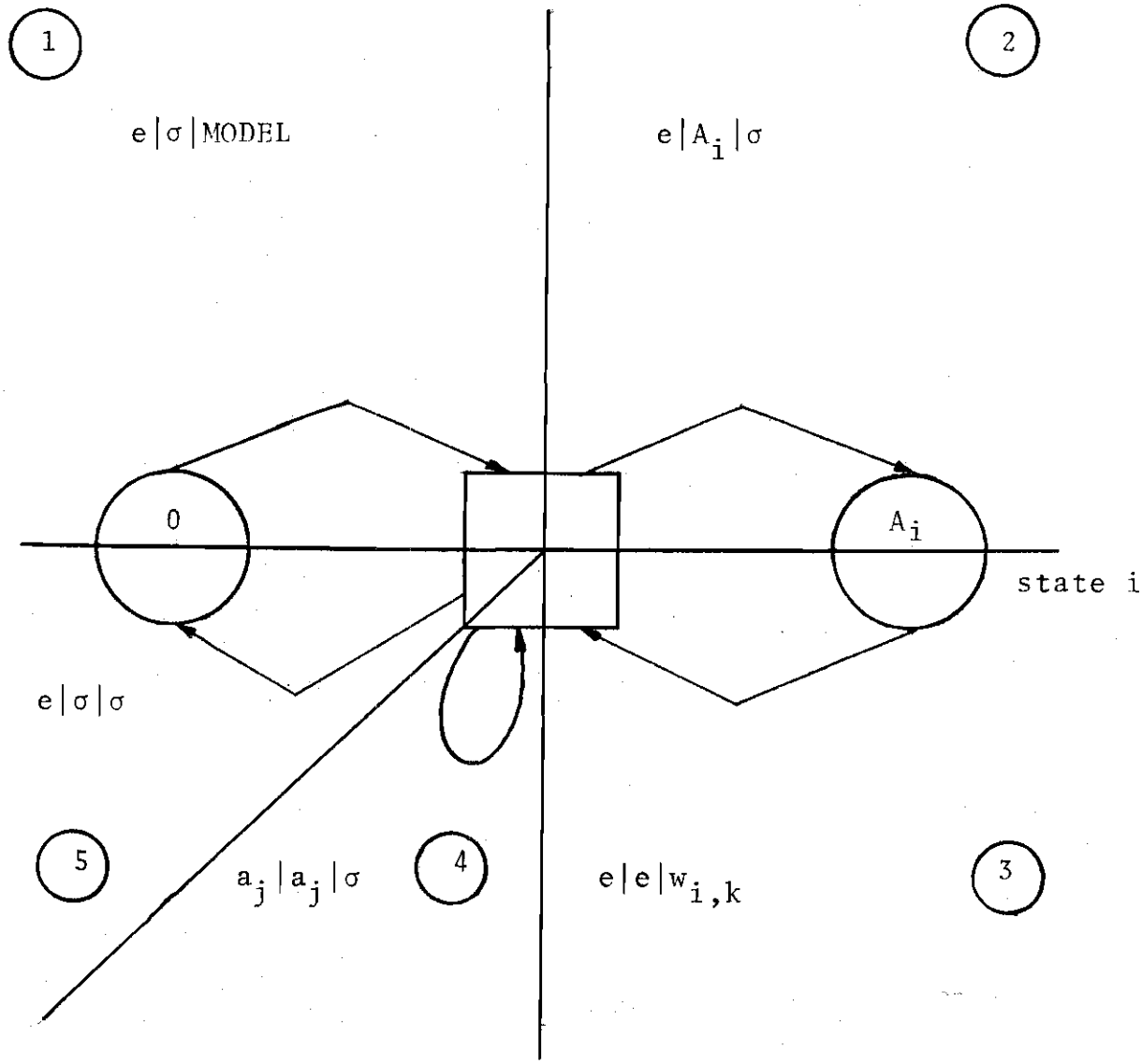


Figure 5. The State Graph of a PDA

1. A grammar is a formal system which generates the set of valid phrases of a language.

2. Operations research (OR) models can be formally described by formal grammars.

3. BNF is a useful meta-language for the description of OR models.

4. The Chomsky system defines four types of grammars with the restrictions on productions defined in a way that they guarantee the inclusiveness of type n grammars into type n-1.

5. Automata are formal systems capable of recognizing languages.

6. An automaton is characterized by its internal states which are images of the syntactical constructs of the language.

7. For each type of grammar there exists an automaton which recognizes the language generated by this grammar.

8. Each automaton can be represented by a state graph where each node is associated with a non-terminal and each arc with a production rule.

9. The overall complexity of each "type n" automaton is substantially lower than the "type n-1". "Type n" is described as the type of automata that recognizes languages of type n.

10. The Chomsky system and the associated theory of automata provide the means of determining and controlling the recognition cost of a specific language.

11. In modifying a language one can estimate the cost-effectiveness of a new feature by observing how this feature affects the structure of the corresponding grammar. It is possible that a new feature can shift the grammar from type n-1 to type n and hence, substantially increase the cost of recognition.

12. A variety of grammars can generate the same language. Thus, grammar transformation is an important part of the design of a cost-effective language.

13. Finite automata recognize exactly the class of linear grammars.

14. Push-down automata recognize exactly the class of context free grammars.

CHAPTER IV

DESIGN AND IMPLEMENTATION OF THE INTERFACING SYSTEM

IV.1. Introduction

The subject of this chapter is the methodology for processing (recognition and interpretation) of languages for operations research (OR) models. These languages as shown in Chapter III are context-free or type 2 in the Chomsky system. The methodology for processing of languages in OR will be presented as a systematic approach. This approach consists of a series of processing phases.

These phases represent a decomposition of the processing into sub-processing phases so that each phase will be responsible for processing certain grammatical constructs of the language in such a way that:

i. the information content will not be altered

ii. a grammatically illegal construct will eventually be detected

iii. the input string will be transformed in a way that will assist the processing by the next coming phase

iv. the system can be modularly designed.

This approach is language independent. However,

depending on the grammatical structure of the language, certain variations could be devised to enhance the economic feasibility of the processing algorithm.

A natural decomposition, used in most compilers, is into the following phases: lexical analysis, syntactical analysis, and semantical analysis.

Lexical analysis is the process in which the basic grammatical constructs (variables, coefficients, etc.) are identified and their grammatical correctness is checked.

Syntactical analysis is the process in which grammatical constructs of higher order (arithmetic expressions, constraints, etc.) are identified and checked.

Semantical analysis is the process in which a meaning is assigned to each construct. This process checks the meaning of a specific construct with respect to other constructs, assuming syntactical correctness. For instance, in a syntactically correct double bounded constraint, the lower bound is greater than the upper bound.

The first two phases represent the recognition process. The last one is the interpretation process. These phases will be presented in more detail in the following sections. Automata as presented in Chapter III will be used as models of recognition algorithms. Also, a state graph approach will be presented. Error recovery and data structures supporting the interfacing system will be discussed.

IV.2. Dialogue Design

The design of the interfacing system should be initiated with the design and analysis of the dialogue of the system. As a first step, the designer of the interfacing system should define the level of communication by identifying various ways of entering the model. The input features should be selected and explicitly stated. At this step, economization should not be a primary selection criterion. Instead, the application environment and the potential user should be considered.

As a second step of the design, the grammatical restrictions on the input should be described. This implies syntactical and semantical specifications. The syntax can be specified by formally describing the grammar. The semantics attached to this grammar should be clearly stated.

The third step of the design should be the analysis of the grammar through formal linguistics, as described in Chapter III of an informal analysis sufficiently detailed to identify the characteristics of the grammar in order to allow an estimation of the expected cost associated with the grammar. This analysis will identify potential transformations and trade-offs of flexibility. The final version of the grammar will be the input to the fourth step.

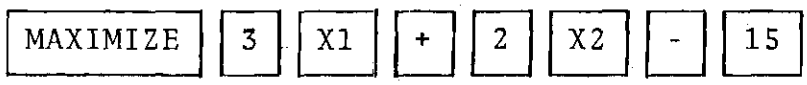
The fourth step of the design should be the selection and design of the supporting software. Algorithms and the mode of their operation for lexical and syntactical analysis

should be defined. The proper cost-effective data structure should also be designed considering user features and the internal OR technique. The data structure is the only part of the interfacing system that will be affected by the OR technique.

IV.3. Lexical Analysis

IV.3.1. Description

The basic function of the lexical analysis is the left-to-right scanning of the input string and the grouping of characters in units which represent the basic classes of the language. The following is the result of the lexical analysis on a typical linear programming objective function:



According to the grammar of the language, these basic syntactic classes (tokens) are non-terminal symbols and hence, they have names. Therefore, X1 is a VARIABLE, + is an ARITHMETIC OPERATOR and so on. Blanks are supressed.

A non-terminal might be mapped to a variety of terminals. Therefore, another function of the lexical analysis is to find the exact mapping element and establish the proper pointers so that the information can be passed to the next phase.

How efficiently the information is processed is a

problem of data structures and it will not be discussed here. For simplicity a simple table is used. This table is usually called the Uniform Symbol Table (UST). It consists of two entries:

i. the class name (non-terminal)

ii. the pointer to the appropriate table

Various support tables will be used by a lexical analyzer for variable names, numeric entries, etc. The UST for the above example will be as follows.

Table 1. The Uniform Symbol Table

Token	Type	Pointer in Tables
MAXIMIZE	OPTIMIZE	1
3	NUMERIC	1
X1	VARIABLE	1
+	ARITHM. OP.	1
2	NUMERIC	2
X2	VARIABLE	2
-	ARITHM. OP.	2
15	NUMERIC	3

This table along with the updated support tables will be the input to the syntactical analysis phase. The support tables are needed to preserve information which will be used by the parser and semantic analyzer. Hence, the lexical

analyzer should correctly update these tables. For instance, if a variable name is scanned by the lexical analyzer, it should be entered in the table for variable names only if it is not already entered.

Through lexical analysis, the information content is preserved and the string is transformed into symbols of unit length, but of higher grammatical order. This will make the syntactical analysis easier since the parser will not be working on characters, but on tokens.

The lexical analysis, more formally defined, is the analysis of an input string through certain linear productions or "linear-like" productions, such as:

<VARIABLE> :: = <LETTER>|<VARIABLE><LETTER>

By scanning the terminal symbols, which are the characters, the non-terminals (tokens) are constructed. At the same time, certain violations of the rules are detected (e.g. illegal characters in an identifier, alphabetic characters in numeric field, etc.).

At the presence of an error the lexical analyzer should be able to recover to:

i. scan the remaining string

ii. create a UST so that the parser will detect any syntactical errors of higher order

This function of the lexical analyzer is called error

recovery and it will be discussed in a later section. The next section discusses the implementation of the lexical analysis.

IV.3.2. Implementation

IV.3.2.a. The Finite Automaton Approach. Lexical analysis works with linear productions or productions that can be handled as linear. For instance, the following production:

<VARIABLE> :: = <LETTER>|<VARIABLE><LETTER>

is linear in the sense that LETTER is defined in terms of terminal symbols <LETTER> :: = A | B | ...| Z. Therefore, for the purpose of lexical analysis we can write:

<VARIABLE> :: = LETTER|<VARIABLE> letter

Assuming that the regularity condition described in Section III.4.1 is satisfied, finite automata (FA) are appropriate models for lexical analyzers. The operation of an FA is based on the state transition function and it is described in Section III.4.1.

In this section, the representation of this operation is described. It will be called functional representation of FA to differentiate it from the graphical representation described in the next section.

It is assumed that the FA is deterministic and statement 2 of Section III.4.1 justifies this assumption. In Aho et al. [2] a method for converting a non-deterministic FA to a deterministic on is presented.

The state transition function M of a FA is a linear function in the sense that it does not require a supporting memory unit. The general format is $M(a_i, q_j) = q_k$ where a_i and q_j are the current input symbol and state respectively and q_k is the next state.

An effective way of representing this function would be by a table where the rows are the q_i and the columns the a_i. Then, the q_k are the elements of the table. For instance, the function of Example 4 in Section III.4.1 would be represented by the Table 2.

Table 2. The State Table of the Grammar of Example 4

a_i \ q_j	MODEL	B	C	D
a	-	F	C	-
b	B	C	-	D
c	-	-	C	-
d	-	-	-	F
BLANK	MODEL	B	C	D

The entry F means that this is a final state and the process has been normally terminated, i.e. the string was

accepted. The last row forces the algorithm to ignore the blanks.

Consider the following grammar specifications whose productions are:

<VARIABLE> :: = <LETTER>|<VARIABLE><LETTER>|<VARIABLE><DIGIT>

<NUMERIC> :: = <INTEGER>|<INTEGER>•<INTEGER>

<INTEGER> :: = <DIGIT>|<INTEGER><DIGIT>

It is observed that the first and third rules can be tabulated. Table 3 could be the corresponding portion of a larger table where the remaining portion of it is represented by the state LINK.

Table 3. Portion of a Grammatical State Table

	LINK	VARIABLE	INTEGER
LETTER	VARIABLE	VARIABLE	VARIABLE
DIGIT	INTEGER	VARIABLE	INTEGER
BLANK	LINK	VARIABLE	INTEGER

However, the second rule could not be handled by the table since it is not linear. With a functional representation of the lexical analyzer this rule will be in the syntax analyzer which has supporting memory units. This implies higher operational cost for this particular rule. In the next section where the rules are interpreted as graphs, it shall be seen that certain nonlinear productions can be

handled by the lexical analyzer. This is due to the fact that each graph has an implicit memory.

However, the tabular approach facilitates the coding and in some cases the speed of analysis. For instance, the prohibited entries of the tables (dashes) could be labels to statements for printing the appropriate error messages. Also, they could contain a linking label for synchronizing the analysis process in the presence of an error.

This tabular simulation would be the framework for the complete lexical analysis. Proper routines for searching and updating tables would be incorporated.

IV.3.2.b. A State Graph Approach. The implementation of the lexical analyzer can be accomplished by coding a state diagram where each state represents a non-terminal symbol of the grammar and each arc represents a terminal symbol. For instance, the productions:

<VARIABLE> :: = letter|<VARIABLE> letter|<VARIABLE> digit

<INTEGER> :: = digit|<INTEGER> digit

can be interpreted as the graph of Figure 6.

The state LINK represents a starting state and the remaining of the grammatical rules. To increase the flexibility, an abstract symbol is introduced. This is the "break" symbol which could be anything that differentiates syntactical constructs including the null. That is to say, the break symbol terminates a syntactical construct and introduces a new one. We obtain the graph of Figure 7.

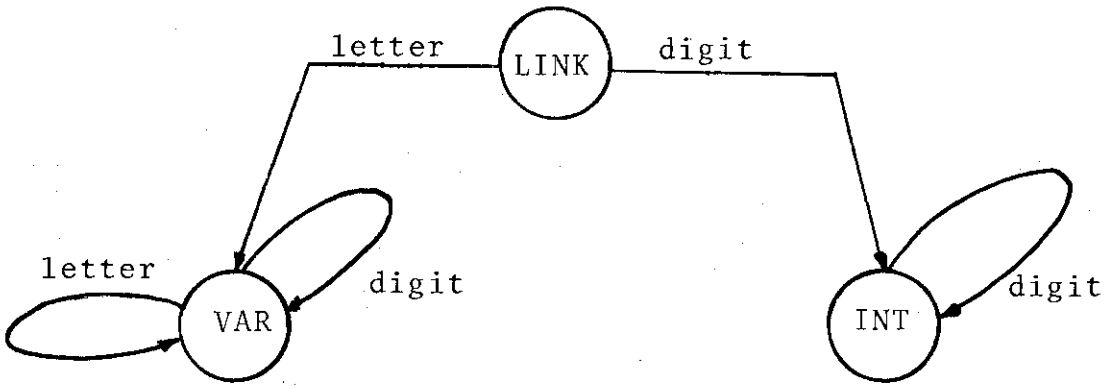


Figure 6. The State Graph of Grammatical Definitions

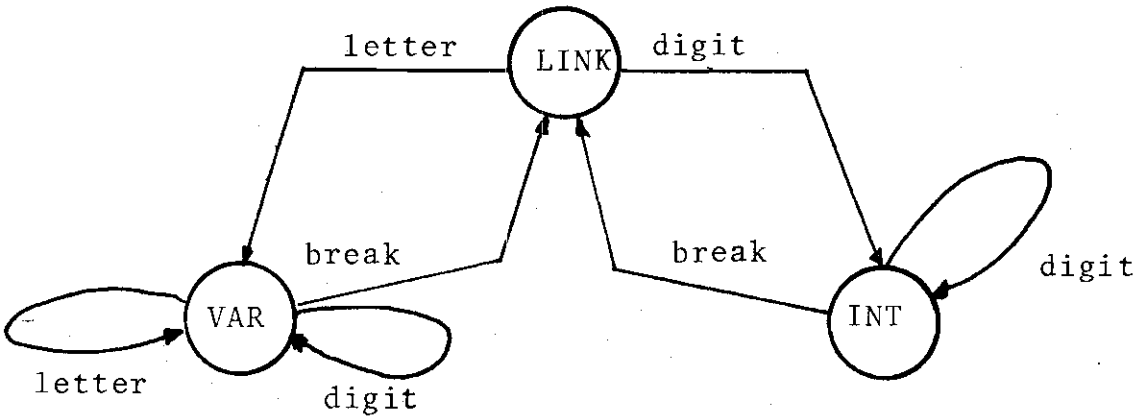


Figure 7. The State Graph of Grammatical Definitions Including Transitions under the Break Symbol

However, the "break" symbol should be used when needed. If one knows exactly where to go from the current state then going back to LINK might duplicate effort. Thus, the "break" symbol should be a symbol which violates the syntactical rules represented by the current location on the graph. This means that the break symbol does not drive the parsing back to the current state nor to any other state connected to the current one. The break symbol causes a transition back to the LINK state where it (the symbol) will either be defined as the starting symbol of another syntactical construct or it will cause an error in the latter case it will be ignored to enable the process to continue.

In Section IV.3.2.a it was discussed that the functional representation does not handle non-linear productions which accrue to the syntax analyzer increasing the operational cost.

The graphical representation should be able to handle certain low-order non-linear productions since there is an implicit memory in the graph. This is illustrated by Figure 8 as a modification of the graph of Figure 7 so that the production <NUMERIC> :: = <INTEGER>|<INTEGER>•<INTEGER> can be handled at the level of the lexical analyzer.

From Figure 8 we observe that in the graphical representation of the 3 productions of Section IV.3.2.a:

i. The construct NUMERIC is represented implicitly by the two states in the square with dotted lines.

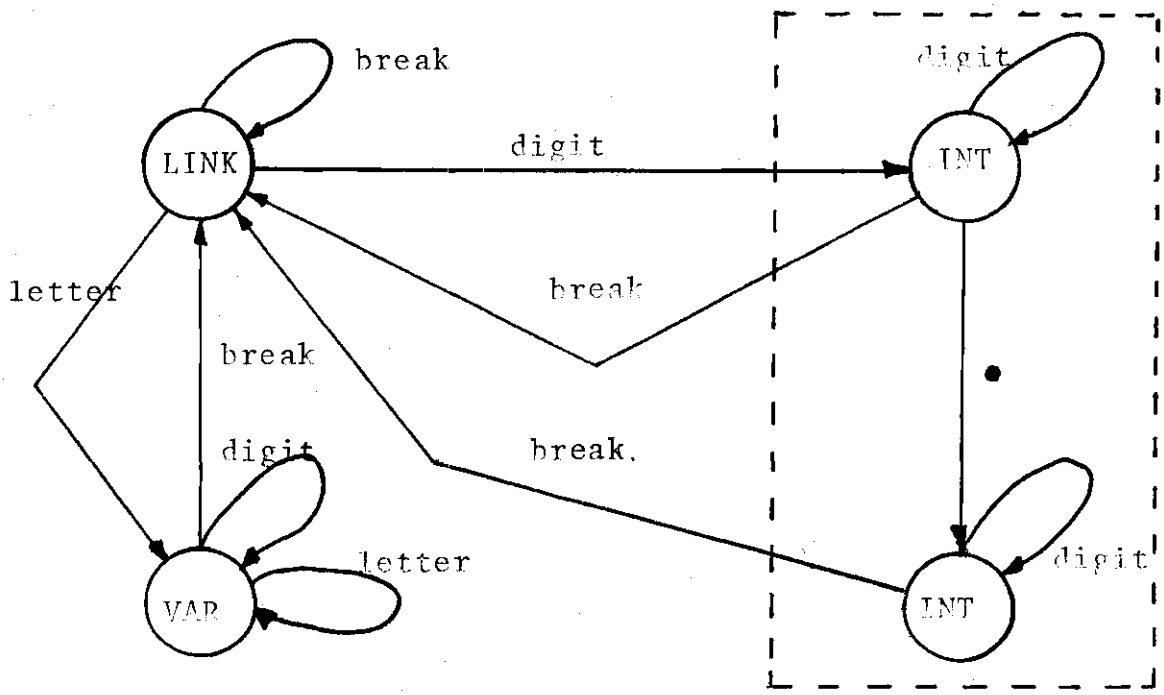


Figure 8. The State Graph of a Grammatical Definition Illustrating the Graphical Representation of Non-Linear Definitions

ii. The "break" arcs are reduced to those needed. As with the functional representation the graphical representation presented above can serve as the framework for a lexical analyzer. For coding, each state of the graph must be represented with a separate portion of code. This approach will make the lexical analyzer highly modular since states and arcs can be easily added or deleted. Hence, grammar modifications can be performed at a low designing cost. Since all the grammatical constructs will be defined on the graph, uncertainty will be decreased, and thus the time of processing will probably decrease in comparison to a table driven parsing.

IV.4. Syntactical Analysis

IV.4.1. Description

The syntactical analyzer operates on the UST (Uniform Symbol Table) which was created by the lexical analyzer. Its basic function is to recognize the syntactical constructs and check the correctness of their construction as specified by the rules of the grammar. At the phase of parsing, the rules which were utilized by the linear analyzer are eliminated from the grammar and the terminal symbols of the grammar at the current stage are the symbols of the UST (first entry). The second entry of the table is not used by the parser, but is used for the interpretation process (semantic analysis) which might be incorporated with the parser.

An error recovery facility should exist in a syntax analyzer for economical reasons. It would enable the parser to locate most of the errors without reprocessing the string. At worst, an error recovery facility enables the parser to locate further errors not related to the first error encountered in a string, while perhaps identifying legal syntactical elements wrongly as errors due to assuming a state after an error which is different from the intended. At best, an error recovery facility whose assumed state matches the intended state sufficiently often in actual practice can indicate specific alternative syntaxes that can be redefined as legal in redesign of the syntax analyzer.

IV.4.2. Implementation

IV.4.2.a. The Push-Down Automata Approach. As was shown in Chapter III, the grammars of operations research models are context-free grammars. Hence, the grammar on which the parser is based is a context-free grammar. A Push-Down Automaton (PDA) will be the appropriate model for the parsing algorithm. The operation of the PDA is discussed in Section III.4.2.

As with the finite automata, we shall represent the PDA through a functional representation. This is to say that the function M of a PDA must be represented. The function M is defined by a set of state transition instructions of the form $s \rightarrow (S_m, x)$ where s is a physical situation, S_m is the next state and x is a string. The physical situation is a triple. So, to store the instructions, a N x 3 table where N is the total number of instructions is needed.

After the function M is defined as above, the representation of the operational rules of the PDA is needed. This is described in Section III.4.2.b. A stack memory unit is required to support the operation.

The PDA as discussed in Chapter III are assumed non-deterministic in the sense that the same left-hand side of an instruction drives to multiple right-hand sides. There are different instructions for each mapping which are translated into separate rows in the functional table. The disadvantage of this is that even if an input string is rejected at a point we cannot say that there is an error until we eliminate all

the alternatives.

Apparently, the operation of a PDA is fairly complex and hence expensive. It is desirable to reduce the number of instructions as well as the uncertainty. This can be accomplished by an extended lexical analysis which will eliminate all the linear and some non-linear productions from the grammar. So it is beneficial to handle at the lexical analysis level all such productions using the PDA to the minimal extent.

After the Uniform Symbol Table is generated by the lexical analyzer, we should redefine the grammar in terms of the symbols in the UST. This usually reduces the non-terminals (states) and the instruction set.

Another point to be made is that having the formal state graph (Figure 5) of the PDA as described in Section III.4.2.d, facilitates the coding, as the lexical relationships can be clearly visualized.

IV.4.2.b. A State Graph Approach. Consider the model:

$$\text{OPTIMIZE} \quad \sum_{i=1}^{N} C_i X_i$$

$$\text{ST} \quad \sum_{i=1}^{N} A_{ij} X_i \geq R_j \qquad j = 1,M$$

The grammar in BNF of the above model was given in Section III.2.

Assuming that the input has been processed by the

lexical analyzer as described in Section IV.3.2, all the linear or "linear-like" productions have been eliminated and the UST has been created. The entries of the UST are the terminal symbols for the reduced grammar of the model. This grammar is of reduced non-linearity since certain non-terminals were converted to terminals. The algebraic analogy to this reduction of non-linearity is the conversion of variables to constants.

Thus, the grammar of the model could be rewritten in its reduced form where entries in small letters denote terminal symbols and the hyphen is used to separate them. Let us call this the "UST grammar." The UST grammar in BNF is as follows:

<MODEL> :: = OPT __<AE>__st __<CONST. SET>

<CONST. SET> :: = <CONST.>|<CONST.><CONST. SET>

<CONST.> :: = <AE> > = num

<AE> :: = sign __num __var | sign __num __var __<AE2>

<AE2> :: = ao __num __var | ao __num __var __<AE2>

where

opt denotes optimize

st denotes subject to

num denotes numeric

var denotes variable

ao denotes arithmetic operator

The operation of a PDA cannot be straightforwardly represented through a state graph as in the case of linear

analyzers. In the latter case a linear production A → xB is converted to a transition from state A to state B through the arc x. At the syntax analysis level, the productions are non-linear with the degree of nonlinearity at least two.

In the previous section, the implicit memory at the state graph was taken advantage of to handle at the lexical analysis level certain non-linear productions. The definition of a non-terminal (i.e. NUMERIC) was decomposed to its elements; each non-terminal was represented by an arc. The same concept can be applied at the syntax analysis level. However, at this level, the productions might have more than one terminal associated with each non-terminal and hence, the terminals cannot be represented by arcs. To solve this problem, a special assumption will be made.

All the symbols of the grammar are represented as states in the graph. As in the case of PDA's, to be able to operate, a neutral symbol (e) will be used to denote the arcs. It is assumed that e preceeds all the symbols (terminals and non-terminals) of the grammar. (The UST grammar is assumed.) The introduction of the neutral symbol (e) can be viewed as a linearization of the grammar. Assuming symbols S_1, S_2, S_3, S_4 the production

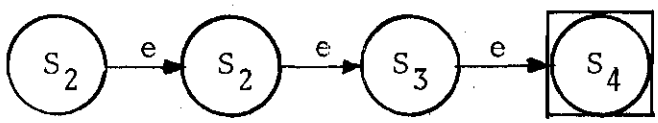
$S_1 \rightarrow S_2\ S_3\ S_4$ is written as

$S_1 \rightarrow eS_2\ eS_3\ eS_4$

The following is a notation where the depth of the structure is illustrated:

$$S_1 \overset{e}{\rightarrow} (S_2 \overset{e}{\rightarrow} (S_3 \overset{e}{\rightarrow} (S_4)))$$

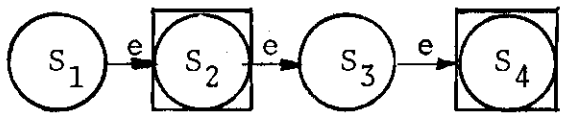
This in state graph notation is:



The S_i's are the states of the graph and the state in the square is the terminal state. There may be more than one terminal state.

The state graph of the production

$S_1 \rightarrow S_2 \mid S_2S_3S_4$ is



Consider the following UST grammar:

$$S_0 \rightarrow S_1 \mid S_0S_1 \mid S_1S_2$$

$$S_2 \rightarrow S_1 \mid S_3S_2$$

The "e-equivalent" of this grammar is

$$S_0 \rightarrow eS_1 \mid eS_0eS_1 \mid eS_1eS_2$$

$$S_2 \rightarrow eS_1 \mid eS_3eS_2$$

The state graph of this grammar is the following:

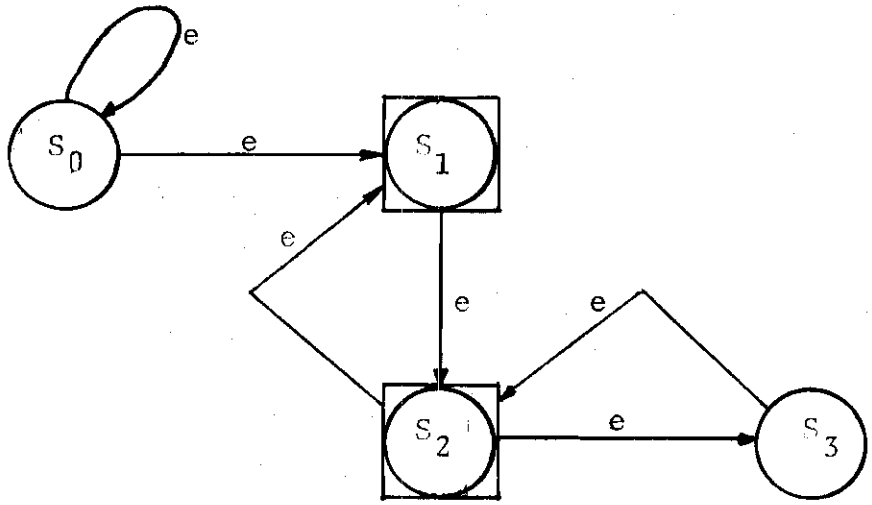


Figure 9. The State Graph of an "e-equivalent" Grammar

Indeed, the string $S_1S_2S_3S_2S_1$ is a legal one since the "flow" reaches the terminal state S_1. However, the string $S_1S_2S_3S_1$ is not legal since there is no path from state S_3 to state S_1.

The amount of memory, implicitly embedded in the above state graph, is exactly the same amount of memory embedded in a 0-1 table representation of the graph as shown in Table 4.

Table 4. The Matrix Equivalent to the State Graph of Figure 9

	S_0	S_1	S_2	S_3
S_0	1	1	0	0
S_1	0	0	1	0
S_2	0	1	0	1
S_3	0	0	1	0

This is to say that the parsing algorithm "remembers" only what is the current and the previous state of passing. This can be considered as the major disadvantage of a table driven passing. It would be desirable to extend the memory of the state graph. This would facilitate features as error recovery or diagnostics. The cause of the problem of limited memory is the recursion through the same state. For instance, in Figure 9 there is recursive flow through states S_1, S_2. A way of extending the memory of the graph would be to consider each state as a family of states, so that state S_j is a type of states and not a single state. Then, all the recursions over state S_j would be eliminated by introducing an additional state of type S_j each time there is a recursive transition over state S_j. That is to say in Figure 9 a new state S_1 and S_2 should be introduced to extend the memory of the graph. Hence, the state graph of Figure 10, where the points indicate an infinite repetition of the same state sub-graphs. The state graph of Figure 10 has an infinite memory.

The following is accomplished by extending the memory of the graph:

i. The locational relationship of each state to other states is explicitly defined.

ii. At each point in time it is known exactly where the parsing flow is.

However, the above approach could be considered as an extreme one. Indeed, this is the case because a closer look will

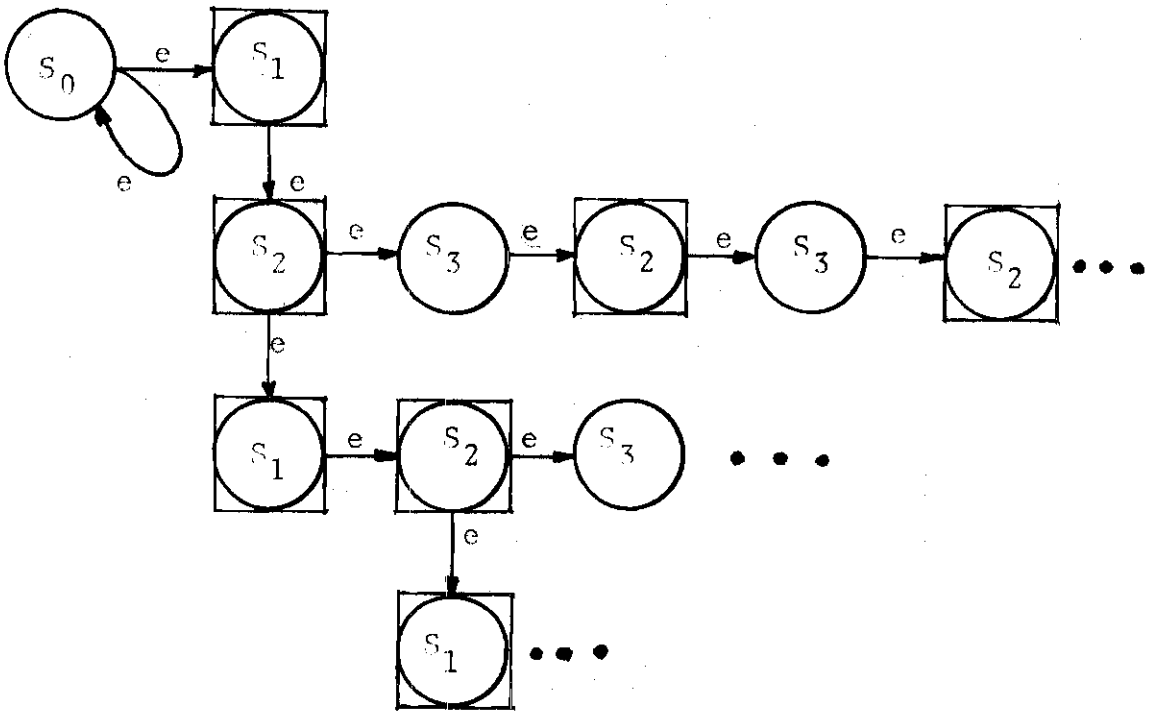


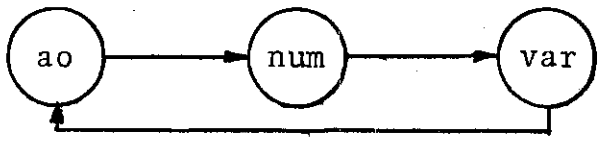
Figure 10. The Extreme Case of Memory Extension of the State Graph of Figure 9

reveal that eliminating all the recursions is redundant. The redundency is due to the fact that certain recursions can be preserved and still have the above two advantages. Certain recursions can be preserved if the following rule is followed: A recursion is eliminated by duplicating states, if the state on which the recursion occurs belongs to two or more syntactical constructs.

Consider the production

<AE> :: = ao__num__var | ao__num__var__<AE>

The corresponding graph is:



The state at which the recursion occurs is (ao). However, it belongs only to the construct AE and the recursion should be preserved.

The above methodology is illustrated with a complete example. Figure 11 is the state graph of the UST grammar presented at the beginning of this section. Figure 12 is the extended memory state graph of the same grammar, but with the recursions eliminated as needed. All the transitions are under the neutral symbol e.

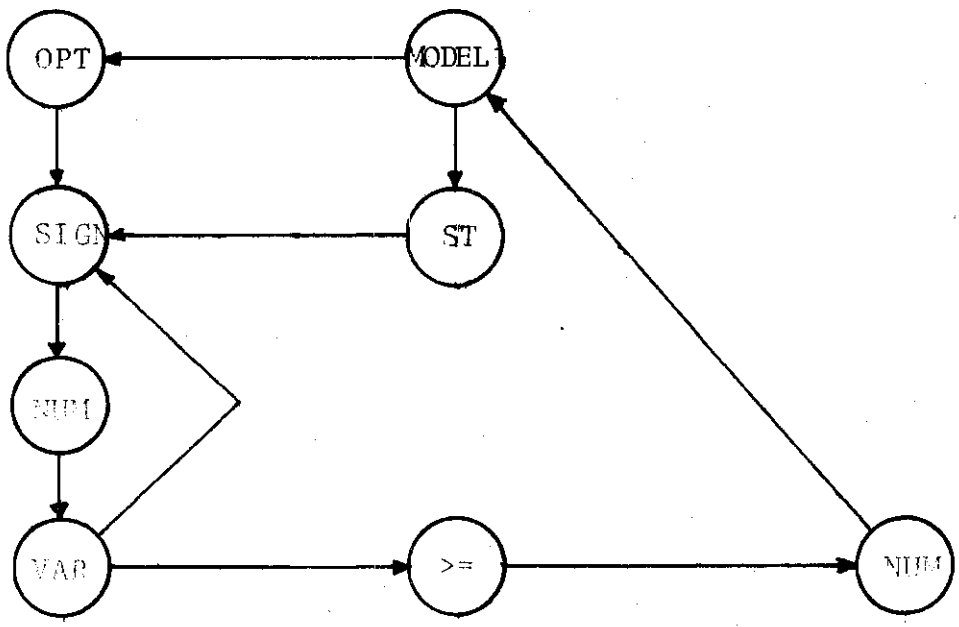


Figure 11. The State Graph of a Grammar of a Simple Linear Model. The memory of the graph has not been extended.

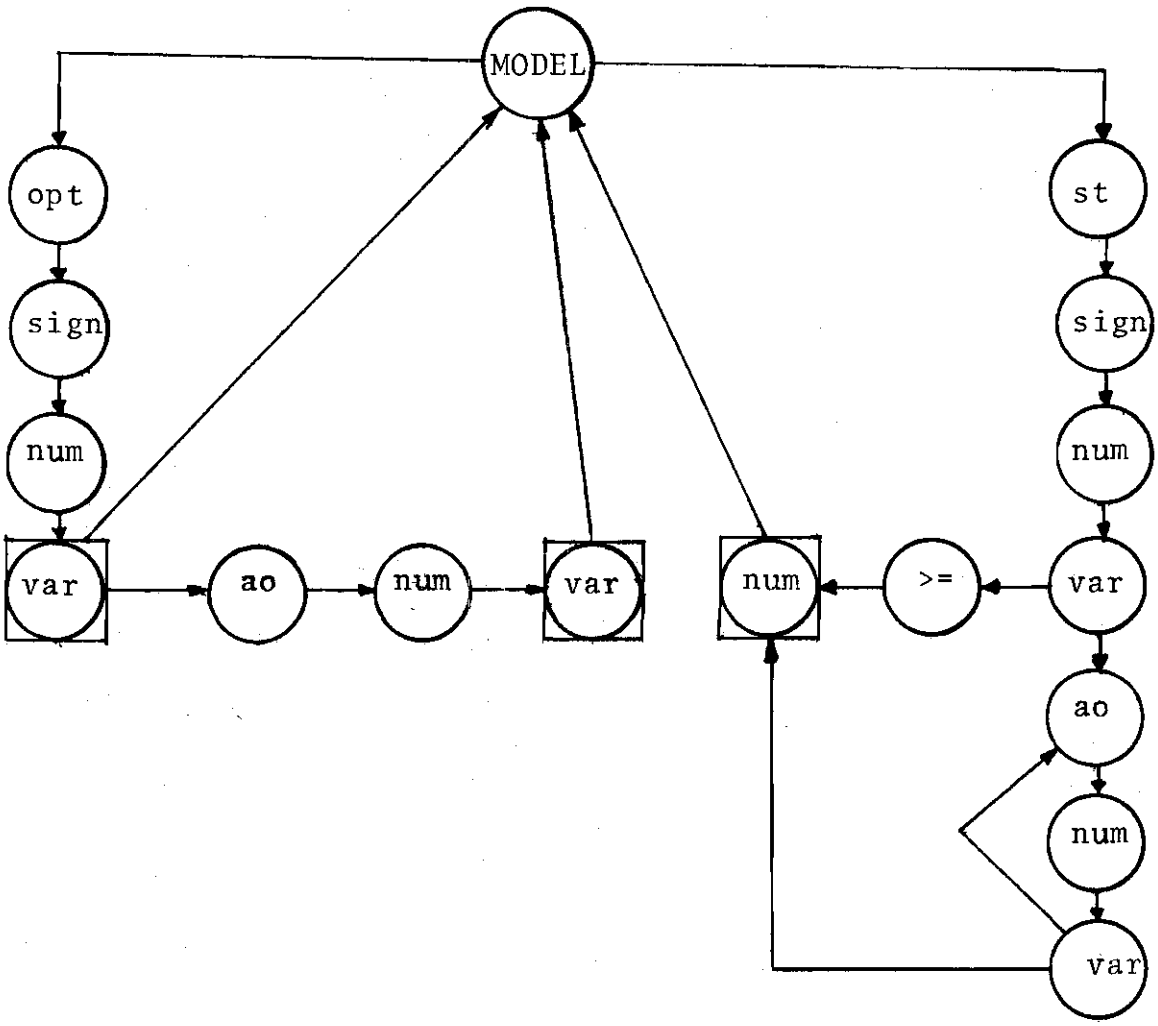


Figure 12. The Partially Extended-Memory State Graph of Figure

In the extended memory state graph certain syntactical constructs are implicitly represented, in the sense that no explicit state was required. These are AF, AE2, CONST. and CONST. SET.

The concept of extended memory can be viewed as an elementary learning process since the more input that is processed the more can be said about the following portion of the input. Hence, this learning increases from left to right. This, combined with the fact that there is prior restriction on the expected input (e.g. linear expressions in linear programming) provides the foundation for efficient error recovery and explicit diagnostics. The efficiency of error recovery will similarly increase from left to right. This is important since the chance of an occurrence of a human error increases from left to right. An additional advantage of the state graph is that any changes in the grammar can be easily implemented by properly modifying the state graph.

This extended memory state graph provides a means for an indirect semantical analysis. Given the type of a problem, there exists some prior knowledge of its syntax at parsing time. If there is a way of explicitly specifying the type of the model, the system could adjust properly the state graph by deactivating certain states.

Let us consider the example of the transportation problem. The corresponding model is a linear one with the restriction that the matrix coefficients be 1. The proper

adjustment of the graph should be the bypassing of the two non-terminal states -num- in the constraint section of Figure 12. If a coefficient different than the implicit 1 is entered, the system will detect this as a "semantical error."

In fact, the restriction of "implicit 1" could be relaxed at the lexical level by supressing all the coefficients equal to 1. This concept could be implemented by represented the characteristics of a specific type of model by a set of active states. Hence, a group of types of linear problems (i.e. networks, machine scheduling etc.) can be represented by a 0-1 matrix where the columns are the states.

A modular design of the system with the proper labeling for the transfer of the execution flow would facilitate the activation-deactivation of the states at a dynamic mode, i.e. at the end of the execution the graph could be reset to its initial configuration.

In addition, the concept of dynamically adjusting the state graph of the system provides a means for detecting certain model characteristics. Given:

i. a set of active states for a specific type of model

ii. an initial configuration of the state graph

it would be possible to determine the type of the model. Moreover, this could be expanded since the linear model, at least grammatically, is considered as a special case (i.e. a

set of active states) of the non-linear model.

The realization of the above concepts requires a sophisticated error analysis process, since there would exist two types of errors:

i. human errors

ii. "model errors"

The state of the art of error recovery leaves no room for high expectations in this area.

The "break" symbol introduced in the state graphs for lexical analyzers is meaningless in the graphs for syntactical analyzers. In an actual design, each state of the graph should be represented by a separate set of computer language statements. This increases the modularity of the system. The length of the code and the coding process appear to be the only disadvantages of the graphical simulation. The advantages of the graphical representation are summarized below:

1. The memory can be expanded up to any desired degree.
2. Certain syntactical constructs are implicitly represented.
3. Efficiency exists with respect to:
   i. grammar expansion or reduction
   ii. error recovery
4. Explicit diagnostics exist
5. Modular design exists
6. No supporting tables are required

IV.5. Semantic Analysis

The semantic analyzer operates on the UST. It concentrates on the second entry of the table. The basic function of this analyzer is to assign the proper meaning to an entry and execute the appropriate information handling.

If a numeric entry is located in front of a variable, it should be interpreted as a coefficient and stored in the proper place in a coefficient matrix. Also, if a numeric entry is interpreted as a bound of a variable it should be stored in the appropriate table reserved for bounds. The semantic analyzer works with the semantic specification of the language. This specification is explicitly specified since one cannot include semantics in the formal description.

The coding of the semantic analyzer usually is an ad hoc procedure. Most often it is implicitly included in the parser. In some cases explicit semantic routines exist. In languages for operation research models, semantic analysis is of considerable importance since it can detect certain irregularities before the optimization routine is activated and computer time is wasted. For instance, bound inconsistencies (i.e. lower bound greater than an upper bound) can be easily detected. However the state of the art of semantic analysis does not provide a theory upon which a semantic routine can be designed. Semantic analysis is performed on an ad-hoc basis either integrated with the syntactical analysis or by a separate semantic routine.

IV.6. Error Recovery

Error recovery is the ability of the system to continue parsing and to detect subsequent errors by either correcting or ignoring the current error.

Referring back to the man-problem environment-machine triangle of communication we can associate three types of errors with the three channels (triangle sides) of communication.

i. Perception errors in the man-problem environment communication channel. These are errors in comprehending the physical and logical characteristics of the problem

ii. Model errors in the problem environment--machine communication channel. These are formulation errors, i.e., the user formulates a model that is not the one intended.

iii. Mechanical errors in man-machine communication. These errors are violations of the grammatical rules of the input language.

The capability for error recovery increases sharply in the same order that the types of errors are presented above.

There is practically nothing that can be done for perception errors. The man-problem environment communication channel is hardware independent in the sense that there is no computer assistance in this communication, and error detection is not possible. Hence, error recovery in this

channel is sole responsibility of the user and it will not be discussed further in this thesis.

There is a small capability of error detection and recovery in the problem environment-machine communication channel. The errors at this stage are formulation errors and as such they are violations of mathematical or logical rules. These mathematical rules most often will be subjective to the particular environment and hense they will be expressible in some explicit or implicit form. This facilitates the error detection and provides the foundation for error recovery. Semantical errors fall in this category of errors. An example of a model error is for a lower bound of a variable to have value greater than the upper bound of the same variable. In this case the recovery strategy could be to reverse the two numerics. However, the state of the art of semantical analysis does not provide the foundation for high expectations for error detection and recovery in this category of errors. For instance there is nothing that can be done if the user enters a numeric entry different than the problem specifications. This could cause a wrong or infeasible or unbounded solution.

There is a substantial increase in the capability for error detection and recovery in the man-machine communication channel. The rules for the man-machine communication are explicitly defined in terms of a grammar and the formal theory of grammars provides a rigorous foundation for analysis.

One of the main advantages of using the formal theory of grammars is that it provides the mathematical framework for error detection and recovery. This is particularly apparent in syntactical errors. Parsing can be viewed as error detection, since it detects the violations of the grammatical rules, and a formal theory of parsing has been developed. However a formal theory of error recovery does not exist and the process is usually designed on an ad-hoc strategy.

According to Young [56] strategies for error recovery can be divided in two classes: Presumptive and non-presumptive strategies. The general approach (i.e. presumptive or non-presumptive) as well as the specific approach (i.e. the actual design of the error recovery routine) are dictated by economic considerations that can be analyzed by benefit profile analysis, as presented in Section V.6. The fundamental economic consideration, for any error recovery strategy, is the trade off between the cost of providing error recovery versus the benefit of minimizing the consequences of a poor or non-existing error recovery. The main consequences of error recovery will emanate from the user's behavior. That is to say to calculate the expected value of the consequences of a specific example we must make probabilistic estimates of user behavior.

A presumptive strategy makes an assumption of state of the point of the error occurrence. This assumption will

be based on previous knowledge, and will be beneficial when there is a high probability of assuming the correct state. This probability varies depending on the location of the error, in the lexical and syntactical respect. That is to say that in some cases a presumptive strategy will be feasible and in some cases it will not be. To illustrate the above point let us consider two typical error cases that occur at the lexical level. The error is the occurrence of a character of a particular type, at a point where it may not supposed (by the grammatical rules) to appear. The decision of whether the approach in each case is acceptable will be based on estimation of the present worth of this approach given by the formula (2) of Section V.6.

i. In alphabetic character appears in a numeric string. A presumptive strategy, in this case, could be to aplace the character by a numeric, i.e. to assume a different parsing state. The constant c of the formula is positive since the user runs the problem on the other hand b (benefit) will be negative since it is worse to get the wrong results and this will happen with probability 9/10 (i.e. 9 out of ten times). Hence the factor (b-c) of the formula (2) will be negative and this will cause the present worth of error recovery of this case, to be negative. This example illustrated a case

where a presumptive strategy would not be desirable.

ii. An illegal character occurs in the string of a key word such as MAXIMIZE or MINIMIZE. A presumptive strategy, in this case, could be to check only for MAX or MIN and to ignore the remaining string until the occurrence of a break character (e.g. a blank). In this case the constant c will be negative while b will be positive and hence the factor (b-c) will be positive. This will cause the present worth to be positive assuming that the initial cost was reasonable.

A presumptive strategy with positive present worth will produce only informatory messages for non-fatal errors. A non-presumptive strategy is for no assumption about the next parsing state to be made.

The incapability of assuming a state is due to the fact we do not have high enough probability of guessing correctly. This would cause a negative present worth even for the best possible guess. An example of a non-presumptive strategy is to ignore the second decimal point when two decimal points appear in the same numeric string. In this case a diagnostic would be generated and the error would be treated as a fatal one (i.e. the problem is not run). In a non-presumptive strategy the problem is not run because there is a small probability that the model is the intended one. Therefore, the expected disbenefit of running a wrong problem

has a much greater expected value than the expected benefit of running a right one. However, the non-presumptive strategy should provide error recovery sufficient for the parser to continue parsing. More on this aspect can be found in Irons [27], and Morgan [35].

Morgan [35] indicates that at the lexical level any error recovery is facilitated by the fact that 80% of misspelling errors fall into one of the following classes

1. One letter wrong
2. One letter missing
3. An extra character inserted
4. Two adjacent characters transposed

At the syntactical level the errors are at higher grammatical order and error recovery is a complex task. Part of the complexity is due to detection of spurious downstream violations, as when not accepting an intended open parenthesis late causes the corresponding closed parenthesis to be spuriously rejected.

Consider the expression

2X1 + 3X2 + X3)

The parser will hold when it reaches the parenthesis, however, the human error is likely to have occurred at a previous point. The central idea behind any error recovery strategy at the syntax level will be one of the following:

1. Delete the trouble symbol and parse again
2. Assume a grammatical state (i.e. insert a symbol) of some point before and parse again. This is similar to backtracking.

In an automaton-based parsing algorithm, this means modifications in the stack. In a state graph approach this means linking to the proper state.

In the environment of mathematical programming there are certain characteristics that relax some of the complexity of error recovery. Most significant is the statements (objective functions, constraints, etc.) are independent of each other as far as the syntax is concerned so the errors are local. That is to say, the errors in one statement do not affect other statements, contrary to the case of compiling general programming languages (i.e. Fortran, etc.).

IV.7. Information Storage and Retrieval

IV.7.1. General Overview

The purpose of this section is not to discuss specific data structures. It is rather to discuss potential approaches which provide a cost-effective information storage and retrieval in the environment of mathematical programming. The cost-effectiveness reflects three considerations:

i. space economy
ii. time economy
iii. cost of modifying the information

These considerations provide the foundation for trade-offs in situations where various options are considered.

In the discussion of the data structures they shall be separated into:

i. static data structures

ii. dynamic data structures.

Static data structures are conventional storage arrangements where the logical order of data is a simple sequential order. Typical static structures are stacks and queues. In a static structure, for instance, the variable names would be stored sequentially in a predimensioned array. Dynamic data structures are storage arrangements where the logical order of the data is established by pointers. Typical dynamic structures are list structures. The mechanism of pointers, which can be anything from a linear list to a non-linear threaded list, is a subject beyond the scope of this thesis and the interested reader should refer to Berztiss [5] or to Brillinger et al. [8].

Static data structures are advantageous with respect to space and time economization, but they can be highly wasteful with respect to the modification of information. For instance, it takes K consecutive computer words to store K items, it takes "zero time" to access the ith element among these K words. However, it takes (k-i) + 1 moves to insert one element between the ith and (i+1)th elements. That cost is multiplied by the second dimension in a 2-dimensional table. So, in a m × n table, it will cost

n(k-i) + n moves to insert a row between the ith and the (i+1)st rows. Complementary to this low-key cost analysis is the corresponding analysis for dynamic structures. With K elements and assuming at least one pointer associated to each element, 2K words are needed. This is the minimum lower bound on space requirements one can have in a list structure.

The access time will be higher since one needs to "look up" at least i pointers in order to access the ith element. In a dynamic structure, the modification of the information and the overall space utilization is much more economical.

With respect to the operations on data, there are:

i. numeric operations

ii. symbolic operations

This separation of operations is necessary since there are numeric data, such as coefficients, right-hand sides, etc., and symbolic data such as variable names, constraint names, etc. An operation is numeric when it treats the data as numbers, such as addition or multiplication. An operation is symbolic when it treats the data as symbols, such as insertion of a new symbol. Static structures are usually more efficient with numeric operations, while dynamic structures are more efficient with symbolic operations. The decision on which data storage and retrieval approach will be taken depends on the third consideration of cost-effectiveness,

i.e., relative importance flexibility to modify the information without reprocessing the input. This seems to be subjective to the specific environment and the importance will be relative to the space resources and time requirements.

In the environment of mathematical programming, space is usually the first in priority and modification is the last. This is correct only if we consider the modifications through the interfacing system. Another potential source of data modifications is the modification at execution time. That is to say, if an optimization process involves internal rearrangements of the data, then this consideration certainly is not the last one in importance.

The data structure attached to an interfacing system must be designed by considering the resources available, requirements in terms of flexibility and the optimization process. This is summarized in Table 5.

Table 5. Data Structure Classification

Storage Scheme	Cost of				Efficiency of	
	Space	Time	Modification		Symbolic Operations	Numeric Operations
			At Input	At Execution		
Static	low	low	Medium	High	Low	High
Dynamic	high	high	Medium	Low	High	Low

IV.7.2. A Data Structure for Linear Programming

The specific environment of mathematical programming with certain characteristics might lead to a data structure that takes advantage of these characteristics. This is illustrated by the following data structure which has certain dynamic features, although it is defined as static since it does not use any explicit pointers.

This data structure stores the coefficients of a linear model in a matrix by direct allocation. It consists of a coefficient matrix and a table where the names of the variables are stored. The variable names are stored with the order of their first appearance in the model. By storing the name of the kth variable, at its first appearance, in the kth location of the table we assign to that variable the order K with no explicit need to store pointers. The coefficient associated with this variable will be stored in the kth column of the coefficient matrix. If a variable name is detected in the input, a table search is initiated to determine if the variable has been assigned an order, or, equivalently, if it has already appeared. If not, it is entered in the table as the last entry and the variable will be assigned that order. If the matrix is initialized with zeroes, the final form of it, after all coefficients have been entered, will represent the entire model.

The disadvantages of the above structure are:

i. The original form of the model is altered with

respect to the sequence of input

ii. A table search must be executed for each variable entry

The last disadvantage should attract special attention since in a typical linear model most of the variables are expected to appear at the beginning of the model. On the other hand, this structure has three major advantages.

i. Prior knowledge of the size of the model is not required

ii. There is no restriction on the order that the variables are entered in an expression

iii. The allocation of the coefficients is final, i.e., no reshuffling is required.

A more formal description of the method is given below. Let

A be the coefficient matrix initialized with zeroes

VIABLE be the table with the variable names

VAR be a variable name

COEF be the associated to VAR coefficient

ORDER (VAR) = I if VAR i in VIABLE (I)

0 otherwise

For the kth row and with N variables currently identified we have:

```
INDEX = ORDER (VAR)
IF INDEX = 0 THEN INDEX = N + 1
VTABLE (INDEX) = VAR
A (K, INDEX) = COEF
```

The above data structure is an example of how the characteristics of an environment can be taken under consideration in the process of selection or design of a data structure. It was illustrated that in the environment of linear programming a static data structure can posess dynamic features.

CHAPTER V

ECONOMIC ANALYSIS OF INTERFACING SYSTEMS

V.1. Economic Attributes

A major difficulty in the area of software economic analysis is the lack of a closed-form metric system. Gilb [23] presents progress towards direction and some of his metrics will be discussed here.

With respect to the interfacing systems, the economic attributes can be divided into two general classes: cost and benefit. This classification will enable a general economic model for system evaluation to be formulated in Section V.3. Figure 13 and Figure 14 illustrate the breakdown of the various attributes. The terminal attributes are the ones that have real economic meaning. The non-terminal ones are used to illustrate the hierarchy and thus clarify the effect of the terminals to the system, so that the analyst can estimate associated costs and benefits.

The terminal attributes will be described and their interrelationship will be discussed. We make the assumption that the number of users has a major economic impact. This will be justified in Section V.3. The following are the cost attributes:

a. PORTABILITY

Gilb [23] indicates that:

"Portability is a reflection of the ease with which a system can be moved from one environment to another"

Portability as defined will affect the number of users.

b. MODULARITY

Gilb [23] indicates that:

"Modularity is the division of the system into substystems. It is a structural property and it affects several other attributes."

Modularity, obtained at some initial cost, will decrease the maintenance and incremental cost while increasing computer time. It will increase the generality and capability for evolution of an improved system.

c. TRANSFORMATION WORK

Transformation work is the energy consumed to peiform a transformation on an input. This includes the input analysis process. For instance, a linear programming model is read, analyzed and transformed to a matrix. The transofmration cost directly affects the operational cost (item f) and it is affected by certain benefit attributes such as free format.

d. TOTAL SYSTEM COST

This is the overall cost of designing or purchasing a system. It is a dependent attribute, i.e., it does

not affect other attributes but is affected by them.

e. INCREMENTAL COST

This is the cost associated with any change of the system. It indirectly affects the number of users while it is directly affected by the modularity of the system.

f. OPERATION COST

The operation cost is the accumulation of the cost of:

i. computer time necessary to enter the model

ii. maintenance cost including the cost for storing the system

iii. cost of the memory occupied during the execution.

Operation cost is affected by the number of users.

g. HUMAN TIME

This is the cost of the time the user has to spend on the terminal to solve his problem, minus a portion of the inactive time, i.e. the time he is waiting for the results.

The following are the benefit terminal attributes.

a. DATA COMPRESSION

This is the ability of the system to accept information in compressed form, e.g. 0-1 VARIABLES or ALL VARS >=0

b. DATA IDENTIFICATION

This is the flexibility of the system to detect and interpret identification codes, such as variable,

constant or constraint names, e.g.

X1 + POWER - C3 X3 = C4

c. GOAL SPECIFICATION

This is the flexibility, with which the user is provided, to explicitly specify the goal of the operation, e.g. maximize, minimize, ignore, etc.

d. SYMBOLIC INPUT

This is the flexibility to use symbols for key words or functions, e.g. SUM XIJ for I=1 and J=1, 3

$$\text{meaning} \quad \sum_{J=1}^{3} X_{1J}$$

e. FREE FORMAT

This is the flexibility to enter the model in real form, e.g., the requirement that allthe variables in a linear model should appear on the left side of the relational operator can be relaxed in a free format input.

f. DETECTION OF CHARACTERISTIC PROPERTIES

This is the ability of the system to detect certain mathematical properties of the model, e.g. linearity, convexity, etc. This feature would be of high value in a universal interfacing system.

g. DATA BANK INTERFACE

This is the ability of the sys tem to build the data bank in the proper format acceptable by a particular

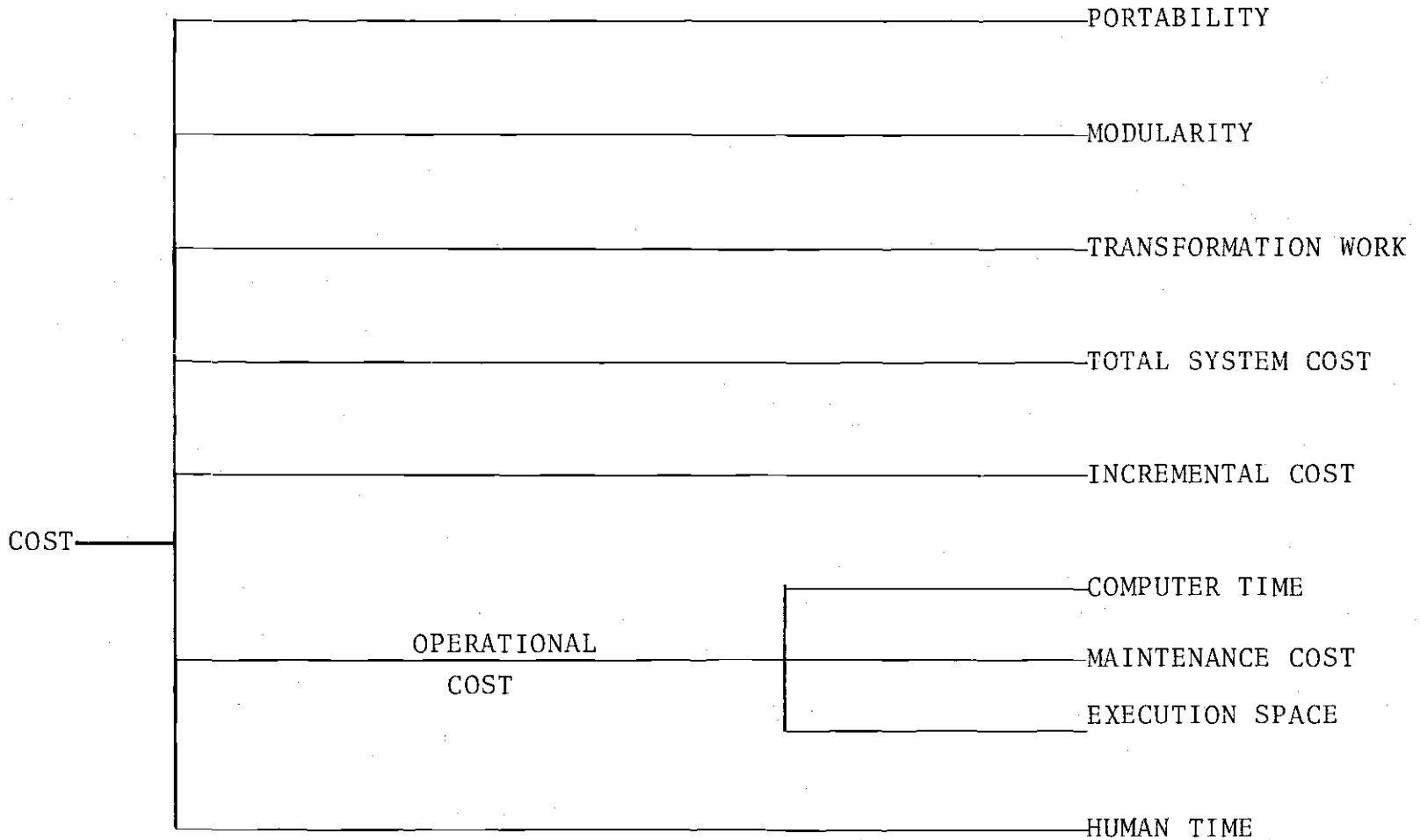


Figure 13. Cost Attributes of an Interfacing System

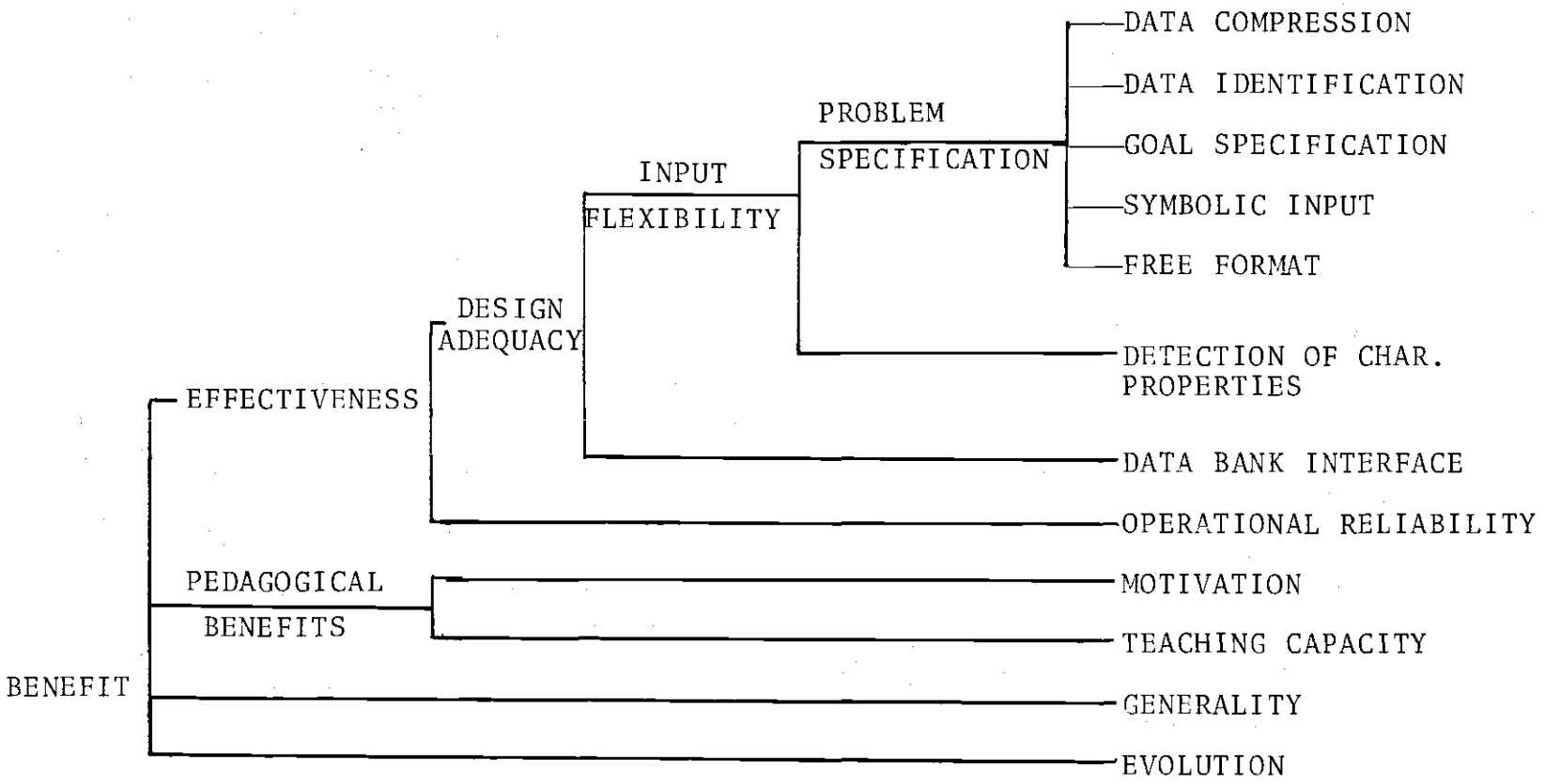


Figure 14. Benefit Attributes of an Interfacing System

canned optimization procedure.

h. Gilb [23] indicated that "Operational reliability is the probability that the system will perform satisfactorily (with no malfunctions) for at least a given time interval, when used under stated conditions"

TRW defined operational reliability as:

$$1 - \frac{\text{number of inputs with execution failures}}{\text{total number of inputs}}$$

i. MOTIVATION

Motivation can be defined as the desire of a user who used the system at least once, to use it again. This attribute directly affects the total utility of the system, and is affected by real-time interactive experiences more than by overall utility.

j. TEACHING CAPACITY

This reflects the instructional capability of the system. The teaching capacity is substantially increased by comprehensive error messages and output. Teaching capacity is very important for systems such as EZLP [28] whose main purpose is pedagogical.

k. GENERALITY

Gilb [23] indicated that "Generality is the degree to which a system is applicable in different environment."

Generality directly affects the number of users.

1. EVOLUTION

Gilb [23] indicates that "Evolution is a designed characteristic of a system development which involves gradual stepwise change." That is to say, "Evolution is the design objective of spreading out necessary changes over a relatively long period of time, so that the rate of change as a function of time is low, or at least lower than alternative design possibilities would give."

The units of the above attributes are in $/X where the unit X must be defined for each attribute. If each attribute is equally "important" then the overall cost or benefit will be the summation of the cost or benefits. If it is not, then "weights of importance" should be designed (with the sum of eight equal to 1) and the overall cost or benefit will be the summation of the products.

V.2. Software Benefit Profile Analysis

In this section, an economic model for estimating the value of a given system with specified characteristics is presented. The same model could be used for sensitivity analysis. An analytical form of the model is an optimization problem where the objective function to be maximized is the present worth of the (benefit-cost) which includes all the "cash flow" over an infinite horizon. In other words, we want to maximize the total economic net gain.

In the case of software economic analysis a potential crucial factor is the expected number of runs per year. In order to formulate the model we must have, or assume, information about the distribution of runs/year over a infinite horizon. For instance, we can assume that the rate of runs/year is normally distributed.

For a computer software system such as EZLP [28], a convenient "benefit profile analysis" for evaluating alternatives can be formulated in terms of the following functions and parameters:

$S(t)$, the "usage profile," a continuous function of time that gives the instantaneous usage rate in runs per year as a function of time;

c, the cost per run, including such things as computer charges and charges for user's time spent in running the software;

b, the benefit per run, perhaps estimated in terms of user's willingness to pay for an alternative to running the software;

c_o, the initial cost of obtaining (or creating, or modifying) the software;

r, the nominal continuous interest rate. $r = \ln(1+i)$, where i is the annual compound interest rate. The one-period present worth discount factor is $1/(1+i)$ or e^{-r}.

For a particular alternative of the software package, an analyst can estimate these functions and parameters and calculate the net present worth for that alternative:

$$P = \int_0^\infty (b-c)S(t)e^{-rt}dt - c_o \qquad (2)$$

A different alternative of the software package would lead to a different estimate of at least one of the entities. Among the alternatives available, the one giving the highest net present worth is chosen. For background on this calculation procedure, see any standard engineering economy textbook. For specific formulas with commonly-encountered functional forms S(t), see Young [55].

A benefit profile analysis can be undertaken on behalf of any set of users. The following examples were provided by Young [56].

Example 1. The designer of EXLP contemplates a modification that will cost $1800 to include. Because of additional overhead, each run is expected to cost $0.20 more than previously, but the typical user is expected to save $0.35 worth of human time per run. The usage profile is estimated as $2000\ e^{-0.03t}$, that is, it is expected that there will be an initial usage rate of about 2000 runs per year, dropping off exponentially at about 3% per year. Let r = 10%. Should this modification be made?

$$\int_0^{\infty} (0.35-0.20)2000e^{-(0.10+0.03)t}\,dt - 1800$$

$$= \frac{300}{0.10+.03} - 1800 = 2307.70 - 1800 = \$507.70$$

Yes, the modification is "worth" about $500 more than its cost.

Example 2. A computer center can receive a copy of EZLP free of charge, getting it up on the system for a total estimated cost of $200 (postage, labor to test it, materials for library documentation, etc.). The probable number of users on the campus is estimated as fluctuating around 80, each using it around twice a year. It will cost $60 per year to maintain and support the system (disc file charges, renewal of documentation, reporting of bugs, protecting against changes in the operating system, etc.). Run cost is ignored, since use of EZLP will displace other pedagogical computer activity of equal cost. In five years, how much would each run have to have benefitted its user in order for implementing EZLP to have been worth its bother, assuming a 12% interest rate (r = ln 1.12 = 0.11329)?

$$P = 0 = \int_0^5 ((b-0)160-60)^{-0.113329t}\,dt - 200$$

b = 1.6967. If the benefit per run is at least this great, the decision to implement EZLP will have been beneficial even if usage stops after 5 years.

Example 3. A sponsor is told by a grantee that the software package he would prepare if the grant were extended would be something for which users would ordinarily be willing to pay $2.50 per run, for five years (after which the package would be obsolete). The number of users (now paying a small license fee) is currently 830 and is dropping off at 15% annually (ln (1-0.15) = -16.2511% annually in nominal continuous terms). Assuming the software package has no effect on run cost, how much of an effect would the software package need to have on the drop-off rate for the willingness-to-pay valuation to justify a $14,000 grant extension? Assume b-c = 3.50 before the inprovement. Let r = 0.10.

$$P = 0 = P_1 + P_2 - 14{,}000$$

where

P_1 is defined as the present worth if the number of users that did not charge

P_2 is defined as the present worth accruing to new users.

Hence,

$$P_1 = \int_0^5 (2.50)(830)e^{-.162511t}e^{-rt}dt$$

$$P_2 = \int_0^5 (6.50)(830)(e^{-gt}-e^{-.162511t})e^{-rt}dt$$

The final form (after the manipulations) of the above equation is:

$$e^{-5(g+0.1)} + 4.43533\ (g+0.1)-1 = 0$$

The solution of this equation is g = -0.051007, i.e. negative drop-off rate. That is to say, an investment of $14,000 changes the software usage from exponentially decreasing to exponentially increasing. An invest of $11,500 would give a zero drop-off rate, i.e. constant software usage.

It is straightforward to generalize equation 1 in several ways, according to the data available. For example, the parameters b and c may be replaced by functions of time without introducing any difficulty. As Young [55] points out, because of the similarity of the integral in equation 1 to the Laplace transform, the properties of the integral for various integrands are extensively known and tabulated.

In the above model the cost and benefit per run should be projected by taking into proper consideration all the associated economic attributes as discussed in Section V.1, i,e., the cost and benefit per run are functions of these economic attributes.

Suppose we assume that linear relationships of the n_1, n_2 economic attributes of cost and benefit (c_i, b_i) exist with w_i, w_i'. The latter are associated with the attribute's weights. The exact type of relationship is subjective to

the particular environment. Linearity is used here to illustrate the mechanism.

Hence,

$$C = \sum_{i=1}^{n_1} w_i c_i \qquad (a)$$

and

$$b = \sum_{i=1}^{n_2} w_i b_i$$

with

$$\sum_{i=1}^{n_1} w_i = \sum_{i=1}^{n_2} w_i' = 1 \qquad (b)$$

The above relationships provide the way for sensitivity analysis under the constraint with respect to the weights. This analysis would be important in studying the sensitivity of the system to a particular attribute. It could be carried out by an analytical method or by some type of simulation. In the case of simulation, an appropriate method could be used if the lower and upper bounds of the C_i's and b_i's were known and a distribution of the values would be assumed or derived. For instance, a Monte Carlo method could be used if a uniform distribution is assumed.

CHAPTER VI

USER PSYCHOLOGY

VI.1. Psychological Considerations

User psychology is an important part of the design of an interfacing system. However, little attention is paid to it, and the mechanics of the system absorb most of the designer resources.

Concentration exclusively on the mechanism of the interfacing can be viewed as a partial design. A complete design should take into consideration the mechanics of the man element of a man-machine system. The objective is not to minimize the present worth of the total cost. It is rather to maximize the present worth of the total net gain. As discussed in Chapter V, the usage rate (runs/year) directly affects this maximization.

User psychology considers the human interface to computer systems. The objective is to identify and analyze behavioral patterns that increase the effectiveness of the system. As such, user psychology falls under human engineering and it deals with rather complex environments. Differences in behavioral patterns cannot be beneficially observed in simple man-machine systems.

Martin [34] indicates that "the differences in capability of men with advanced computer dialogues are as

great as their differences in capability at a piano keyboard. But with the simplest dialogues all men are equal."

User psychology provides the designer with information about the potential user, i.e. capabilities, intelligence, amount of time acceptable to learn a syntax, psychological effects of the design (motivation, boredom, etc.). The overall effectiveness of the system will be increased if this information is properly utilized during the designing process.

Computers are not as intelligent as many people think they are. There are "intelligence gaps" and this becomes more apparent in the problem solving area. A successful man-machine system should utilize that available on the terminal, human intelligence and fill the gaps of the machine. This would also substantially contribute to the effectiveness of the system. Hence, the interfacing system should have the capabilities of retrieving from the user not only information but intelligence as well. User psychology provides an insight in this area.

Martin [34], indicates that the user psychology should be studied at three levels. The first one is the functional considerations. It studies the distribution of functions to the man and to the machine, i.e. which functions should the user perform and which should the machine perform. It is at this level that human intelligence will be utilized. One important consideration of this level is the ability of the interfacing system to retrieve human experience which cannot

be quantified and programmed explicitly in the machine. This is very important in the environment of operations research where one "channel of flow" of human experience and intelligence is the mathematical model. The capacity of this channel is increased if the interfacing system provides a high degree of input flexibility.

Human judgement and experience is further effectively utilized if system provides editing capabilities. In the OR environment it is a functional consideration of user psychology to decide whether the model will be formulated by the user or the machine. In other words, the user could enter an explicit formulation or certain key characteristics and objectives of his problem.

The second level is that of procedural considerations. Having decided the distribution of functions between man and machine, the operation must be organized in a sequence of procedures. When is the model entered? When are error messages displayed? When can the user edit the input? These are questions to be answered at the procedural level. A good design should avoid the "human channel overload," or user's boredom. On the other hand attention should be paid into creating to the user motivation for using the system. For instance, motivation can be increased by a comprehensive output. With respect to user's boredom, an important procedural consideration is the flexibility of the system to accept input through alternative sequences of instruction.

Experience indicates that the user becomes bored by a system that requires a fixed input sequence, especially if the length of it is not controlled by the user. He should be able to effectively utilize his knowledge of the system.

The third level is that of syntactical considerations. This mainly deals with the effect of the syntax on the user. Experimental psychology provides insight into how a particular syntactical feature is accepted by the user. The syntax can affect:

(i) the time necessary for a system to be learned,

(ii) the chance of an error occurrence.

Through statistical investigation researchers have concluded that certain conventions of natural language should not be used. For instance, abbreviations as ISN'T or DON'T should be written as IS NOT or DO NOT. Also words as ILLEGAL or UNKNOWN should be written as NOT LEGAL or NOT KNOWN.

During the process of research for this thesis, three systems were designed. These are EZLP [28], NLP (non-linear programming) [39] and the automatic Cut Order Planning System for the apparel industry [32]. These systems will be discussed in the applications chapter. EZLP was the main carrier of this research and it will be discussed in a considerable detail.

With respect to user psychology, the experience from the EZLP usage in various departments at Georgia Institute of Technology, is summarized as follows:

1. The requirement for familiarizing with the input language was substantially relaxed. Students expressed their satisfaction that, due to the free format, they practically did not have to invest time in learning the system.
2. Input flexibility combined with a variety of simplex-based methods, included in the system, created motivation for use and decreased the boredom from perpetual usages.
3. Explicit diagnostics resulting from an effective error detection and recovery routine decreased the time the user has to invest per problem.

An experimental fact with use of EZLP and similar programs having a user-oriented input language is that the user very quickly learns to bridge the communications gap and passes from a state of fear and desire for a problem-oriented language to a state of impatience and desire for minimizing total keystrokes. This rarely takes more than three runs with EZLP. After that, the main concern becomes that of insuring that input already prepared is not lost.

With respect to NLP the experience was that the use of Fortran syntax for the mathematical model was very well accepted by the users. Fortran was selected as the input language since, at the time of the NLP development, Fortran was the most widely used and known computer language. The selection of Fortran as the input language provided to the

user a powerful parser (Fortran compiler) with efficient error detection and recovery features. A primary goal in the design of NLP was the minimization of the cost of operation.

The human typing effort was considered to be the most significant cost. To accomplish this minimization of cost several functional and procedural activities were made implicit requiring no action on the part of the user.

The experience with the automatic Cut Order Planning system was that the productivity and creativity of the user was increased. This was accomplished by relaxing the user from formulating and entering the mathematical model for the production planning. Instead, the system provided to the user the flexibility to intelligently formate and enter the specifications of the problem.

The above experimental results indicate that, with respect to the user psychology, the designer should consider the following.

1. Minimize the time necessary for the user to become familiar with the system.
2. The design should create to the user motivation for using the system.
3. The design should try to minimize the chance for user frustration due to diagnostics or other system behavior.
4. The human time on the keyboard should be minimized.

5. The user's creativity and experience with the problem environment should be effectively utilized.

VI.2. Human Short Term Memory and Attention Channel

An understanding of the human short term memory and attention channel might have a substantial effect on the design of the system. This is due to the fact that the human buffer and channel are of limited capacity. Psychologists have attempted to experimentally measure these capacities and they have gained considerable insight into the operation of the buffer and the channel.

With respect to the attention channel capacity, results indicate that the user of a terminal should not be expected to differentiate between more than seven classes of unidimensional information. On the other hand seven is considered to be a low number when dealing with problem solving. Fortunately, from experimentation, it is known that there are several ways to increase the channel capacity at interaction, i.e. the interface system can be designed in a way that the user's channel capacity will be temporarily increased. A discussion of these channel expansion methods is provided by Martin [34].

Interestingly enough, the human short term memory has a capacity of seven items of information. The brain can store up to seven different perceived stimuli for immediate

use. A class of information has a capacity of its own, and experimental evidence shows that the human buffer can handle as much information, of the same class, as possible.

Martin [34] indicates that "In other words we must plan the thinking process in such way that the short-term memory deals with chunks that are identifiable as a single item but that contain as much information as possible." The process at organizing information in general classes is called "recoding."

Martin [34] indicates that "It is clear that in the future recoding will play a major role in man-computer dialogues." These experimental results support some real life experiences. For instance we more easily memorize for immediate use a telephone number if part of it is "recorded" to a name. Hence we can memorize, for immediate use more telephone numbers.

Consider the touchtone telephone user, who memorizes, in the short term memory, patterns of pressing buttons instead of numbers. Each pattern is an item of information and hence the user can memorize at most patterns. Also consider the user who records telephone numbers into words and then the words into known titles of books or movies. He can hold in his buffer at most seven titles, but he has considerably increased his capacity in terms of telephone numbers.

This concept, should be taken into consideration when

designing interactive system. Several of the economic attributes could be considerably affected.

CHAPTER VII

APPLICATIONS

VII.1. Linear Programming

VII.1.1. EZLP

EZLP is an interactive computer system for linear programming problems. EZLP was developed in the School of Industrial and Systems Engineering of Georgia Institute of Technology [28]. The goal of the system was to increase the computer share in formulating and solving a linear programming problem. In that respect EZLP is a step towards narrowing the gap between the OR analyst and the computer in Figure 2.

In accordance with the principle of balanced man-machine communication, EZLP was designed to reduce the user's transformation work by increasing the share of the computer of this transformation work. In this respect the system provides certain facilities not found in the existing similar systems.

An informal characterization of the EZLP syntax is that it provides a free formation with respect to the mathematical model. Hence, all the restrictions of a typical LP system are relaxed. Usual restrictions are:

i. The linear expression must be entered before the relational operator. This implies that, usually, only a numeric entry is allowed to the

right of the relational operator.

ii. The variables should be entered in the same sequence.

iii. The dimensions of the problem should be specified in advance.

EZLP relaxes the above restrictions and provides certain additional features as summarized below:

1. There is no restriction to the number of alternative objective functions.

2. Single variable constraints with the same right hand side can be grouped into a single constraint (explicit list constraint) e.g. X1, X2, X3 >= 10

3. A bound can be assigned to all the variables e.g. ALL VARS >= 5

4. A bound can be assigned to all the variables not assigned a bound up to the point of entrance of the constraint e.g. ALL OTHER VARS >= 10

5. Different bounds can be assigned to the same variable in different points in the model but the minimum upper bound and the maximum lower bound will prevail.

6. Arithmetic expressions can be bounded from above and below e.g. and 5 = X1 + X2 + X3 <= 10

7. Arithmetic expressions can be entered in both sides of a relational operator, e.g. and X1 + X2 <= 10 - X3

8. There is no restriction to the order of variables e.g. and X5 + X3 + X4 <= 10 - X2 + X1

9. Indexed variables are allowed, e.g. X1,2,3 X25,3

10. Numeric values can be entered in an arithmetic expression as single arithmetic entities. The summation of all these numerics will form the right hand side of the constraint, e.g. and ROW3: 3 POWER -2 + 6 HEAT + 10 <= 15 it is equivalent to

and Row3: 3 POWER + 6 HEAT <= 7

11. Prior knowledge of the size of the model is not required.

The design of the interface of EZLP was based on the methodology presented in the previous chapters. The EZLP grammar is a context free one with high degree and high density of nonlinearity. As such, a push-down automata approach would be expensive time and space wise. Instead, the state graph approach was used to perform lexical and syntactical analysis. The complexity of the EZLP grammar was such that the design took advantage of all the advantages of the state graph. Memory extension and implicit representation of states increased the time efficiency and the EZLP recognition is a fast process.

Semantical analysis, embedded in the process of syntactical analysis is limited in checking the consistency of bounds of the variables and expressions.

Error recovery assists the continuation of parsing. Linearity, represented by a rather not complex grammar, provides the foundation for efficient error recovery

without the support of error correction. EZLP detects most of the errors in a line. However, in certain occasions it stops at the first error ignoring the remaining part of the input sting. This occurs when it is suspected that several artificial errors would be generated by error propagation should the scanning process continue. The state graph in conjunction with linearity provide the facility of pointing the exact error point with a self explanatory diagnostic message.

The data structure presented in Section IV.7.2., with several support tables for bounds, variables, etc., was used to store the information about the model.

The following is the EZLP CF grammar in BNF:

Notation:

IMPL. LIST CONS.	denotes	Implicit List Constraints
EXPL. LIST CONS.	denotes	Explicit List Constraints
ALT. OBJ. FCN	denotes	Alternative Objective Function
AE	denotes	Arithmetic Expression
INT. AE	denotes	Internal Arithm. Expression
REL. OP	denotes	Relational Operator
AO	denotes	Arithmetic Operator
NUM	denotes	Numeric
VAR	denotes	Variable
SPECS	denotes	Specifications
COEF	denotes	Coefficient
LP	denotes	Linear Programming

Definitions:

<LP MODEL> :: = <OPT><LINE NAME><AE><MODEL SPECS>

<MODEL SPECS> :: = <MODEL ENTRY>|<MODEL ENTRY><MODEL SPECS>

<MODEL ENTRY> :: = <ARITHM. CONSTRAINT>|<LIST CONSTRAINT>|
<ALT. OBJ. FUN>

<LIST CONSTRAINT> :: = <EXPL. LIST CONS>|<IMPL. LIST CONS>

<ARITHM. CONSTRAINT> :: = AND <LINE NAME><AE><REL. OP><AE>|
AND <LINE NAME><NUM><DEL. OP.><AE>
<REL. OP>

<ALT. OBJ. FUN> :: = ALSO <LINE NAME><AE>

<EXPL. LIST CONS> :: = AND <LINE NAME><NUM><REL. OP.><VAR LIST>|
AND <LINE NAME><VAR LIST><REL. OP><NUM>|
AND <LINE NAME><NUM><REL. OP><VAR. LIST><REL. OP.><NUM>

<IMPL. LIST CONS.> :: = AND <LINE NAME> ALL <OPTION 1> VARS
<OPTION 2>

<OPTION 1> :: = OTHER|null

<OPTION 2> :: =<REL. OP><NUM>|URS

<AE> :: = <SIGN><TERM>|<SIGN><TERM><INT. AE>

<INT. AE> :: =<AO><TERM>|<AO><TERM><INT. AE>

<TERM> :: = <COEF><VAR>|<NUM>

<VAR LIST> :: = <VAR>|<VAR >, <VAR LIST>

<VAR> :: = <LETTER>|<LETTER><VAR STRING>

<VAR STRING> :: = <PREFIX STRING>|<PREFIX STRING><VAR STRING>

<PREFIX STRING> :: = <DIGIT>|<LETTER><INDEX>

<INDEX> :: = <DIGIT>, <DIGIT>|<INDEX>, <DIGIT>

<LINE NAME> :: = <VAR>| Null

<COEF> :: = <NUM>|Null

<NUM> :: = <INTEGER>|<INTEGER>|<INTEGER>.|
<INTEGER>•<INTEGER>

<INTEGER> :: = <DIGIT>|<DIGIT><INTEGER>

<DIGIT> :: = 0|1|2| --- |9

<LETTER> :: = A|B| --- |Z

<OPT> :: = MAX | MIN | MAXIMIZE | MINIMIZE

<SIGN> :: = + | - | Null

<AO> :: = + | -

<REL. OP> :: = <= | = |> = | =<| = >

The above grammar after the lexical analysis is transformed to the following UST grammar.

<LP MODEL> :: = Opt. - Line name - <AE><MODEL SPECS>

<MODEL SPECS> :: = <MODEL ENTRY>|<MODEL ENTRY><MODEL SPECS>

<MODEL ENTRY> :: = <ARITHM. CONSTRAINT>|<LIST CONSTRAINT>|
<ALT. OBJ. FUN>

<LIST CONSTRAINT> :: = <EX PL. LIST CONS.><IMPL. LIST CONS.>

<ARITHM. CONSTRAINT> :: = AND - Line name - <AE> Rel. op - <AE>|
AND - Line name - num - rel. op -
<AE> - rel. op. - num

<ALT. OBJ. FUN> :: = ALSO - line name - <AE>

<EXPL. LIST CONS.> :: = AND - line name - num - rel. op -
<VAR LIST>|
AND - line name - <VAR LIST> -
Rel. op. - num |
AND - line name - num - rel. op. -
<VAR LIST>| - rel. op - num

<IMPL. LIST CONS.> :: = AND - line name - ALL - <OPTION 1> -
VARS - <OPTION 2>

<OPTION 1> :: = OTHER | Null

<OPTION 2>:: = rel. op - num URS

<AE> :: = Sign - Term | Sign - <TERM><INT. AE>

<TERM> :: = Coef - Var | num

<VAR LIST> :: = Var | Var, - <VAR LIST>

The state graph corresponding to the above grammar is shown in Figure 15. In this graph certain non terminals are represented implicitly, i.e. by a group of states. These are all the non-terminals at the left of the definitions.

The terminals - sign - and - coef - are omitted since their difference from their counterparts - ao - and -num- is the null element.

The EZLP grammar could be dynamically adjusted, at low cost as described in Section IV.4.26. So, special forms of linear programming could be inputted through the EZLP interface with the error recovery mechanism reflecting their special characteristics for instance, if EZLP was to be adjusted to handle only models with 0-1 coefficients, the only change should be to redefine TERM as <TERM> :: = <VAR>|<NUM>.

This in the state graph translates into eliminating the arcs leading from the state num to the state var. Then the generic state graph of AE (Arithmetic Expression) should be as in Figure 16.

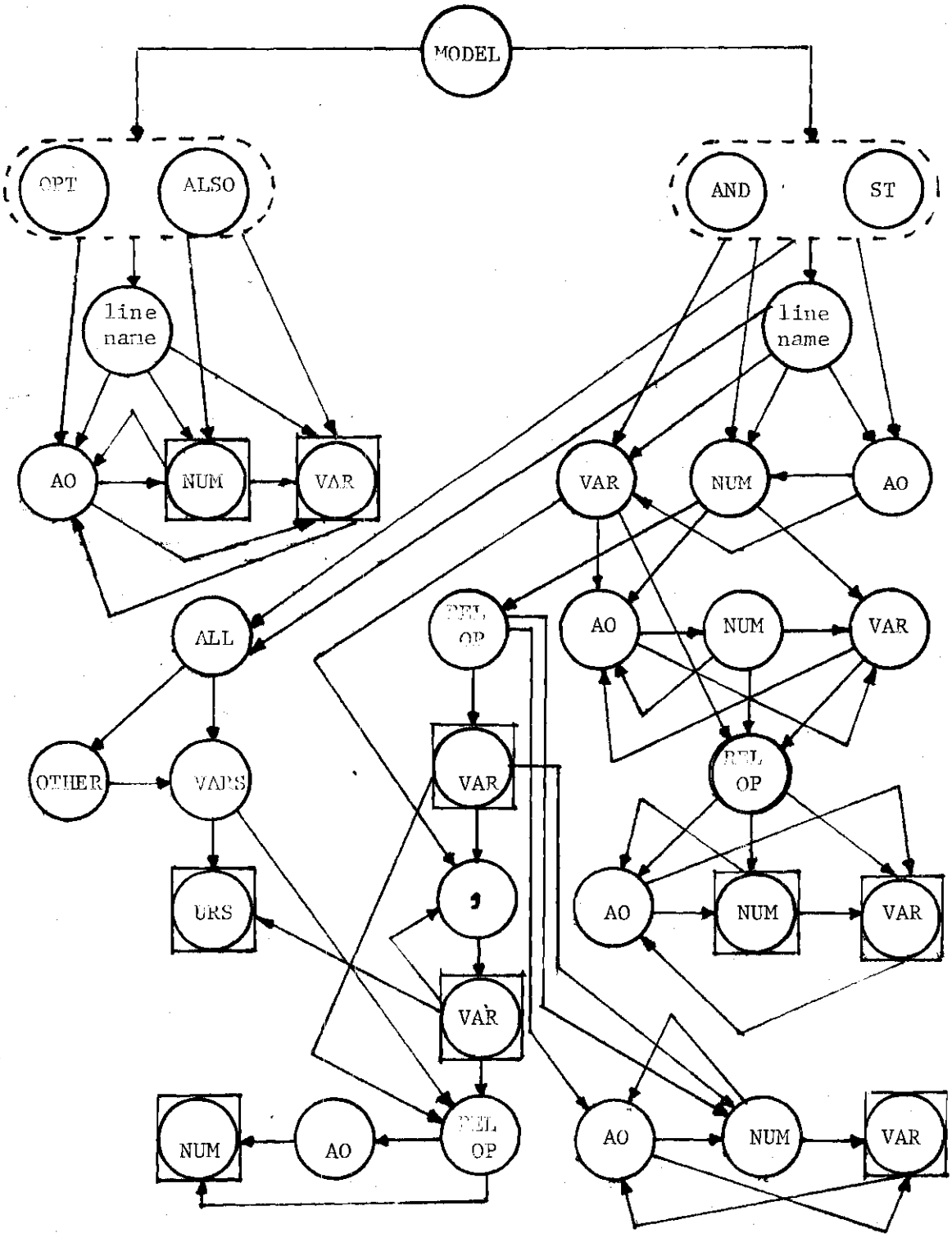


Figure 15. The State Graph of the EZLP Grammar

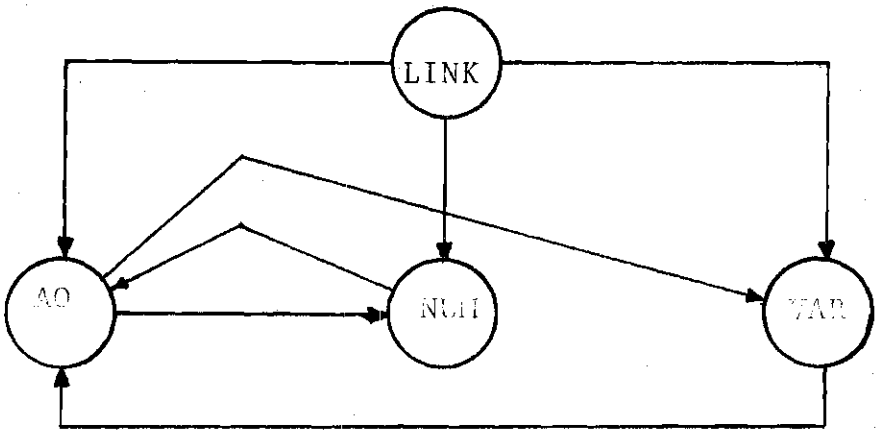


Figure 16. The State Graph of an Arithmetic Expression with 0-1 Coefficients

Aspects of user psychology were considered in the design of EZLP. The high flexibility of input reduce the user's boredom and increases the matiration to use the system, especially in the academic environment. The capacity of the user's attention channel has been increased by increasing the information capacity of the EZLP entries. Constraints as AND X1, X2, X3 >= 10 or AND ALL OTHER VARS <= 5 are examples of entries with increased information capacity, for the same reason, the capacity of the user's short term memory has been increased. As a result, human judgement can be exercised on a more effective way in the process of formulating or editing the model.

VII.1.2. Higher Level Linear Models

By comparing EZLP to the other existing interfacing systems for LP it is apparent that there are mainly two advances:

i. elevation of the communication closer to the user's thinking level

ii. higher rate of communication

The first can be considered as a contribution to the task of "retrieving user's experience" and hence a more effective man-machine system. The second is a major economic consideration affecting several economic attributes.

EZLP and its variations represent a man-machine communication at a fixed level of information compression. This implies that the rate of communication of the EZLP variation will be approximately the same, i.e., a grammar transformation will not substantially affect the rate of communication. On the other hand it may affect the "experience retrieval."

To substantially increase the rate of communication in the linear environment the system should provide facilities of symbolic input of higher information capacity. For instance the arithmetic expression

MAX C1X1 + C2X2 + ... + C10X10 could be inputted as:

MAX SUM C1X1 I = 1, 10 or

MAX SUM C1X1 I = 1,2,3,...,10

Note the simplicity of the grammatical definitions:

<OBJ FUN> :: = <OPT> SUM <COEF> X <INDEX><INDEX> = <NUM>, <NUM>

<COEF> :: = <INDEX>|Null

These facilities, if carefully designed, considering the potential user psychology, may substantially affect the

rate of communication and the extraction of information from the user. A system communicating with the user at a higher level is the Markamatic Cut Order Planning (MCOP) developed by Camsco, Inc. in cooperation with the School of Textile Engineering of Georgia Institute of Technology [28]. MCOP is utilized for production scheduling in the apparel industry. The operation is based in a close cooperation between the user and the machine with the support of intelligent terminals. The goal of the system is to find the most economic utilization of fabric given a set of orders to be satisfied. The process is decomposed into two optimization phases. The first phase is performed by the human who allocates, via a graphics terminal, garments, usually of irregular shapes, into rectangular areas. If the grouping which is called marker of the garments causes an acceptable fabric utilization, it is given an identification code and it is stored in the computer. The associated percentage of fabric utilization is also stored. This process is repeated until all the requirements are satisfied, i.e., until all the garments for production are allocated into markers.

The output of the first optimization phase is processed by various subsystems which formulate a linear mathematical model. The variables at the model are the markers from the first phase.

A simplex based integer optimization process is activated and the output is the markers and the quantity of

those markers to be produced. A final modulus transforms the solution into a production form readable by the production personnel.

The above system is a step closer to the problem environment. It allows the user to communicate with the computer at the level of problem specifications, relaxing him from the formulation of the mathematical model. Also, by decomposing the optimization into two phases, the user is given the opportunity to effectively utilize his experience and intuition. The system is monitored by a small operating system consisting of two character commands. MCOP is an illustrative example of how a system can cooperate with the human in the problem environment at a high rate of communication.

VII.2. Non-Linear Programming

The concepts discussed in the previous chapters apply to processing non linear models. Once the features of the system have been decided the grammar should be specified in an efficient metalanguage. This specification for non-linearity would be similar to the general linear model specification with the addition of non-linear arithmetic expressions.

As an example the grammar of EZLP of the previous section can be easily converted to a grammar for non-linear input. All the features of EZLP would be preserved if AE,

INT. AE and TERM are the only redefined nonterminals. The new definition should be a broader one able to accept non-linear entries as well as linear. This could be accomplished by the following definitions.

<AE> :: = <SIGN><TERM>|<SIGN><TERM><INT. AE>
<SIGN><NUM>|<SIGN><NUM><TERM><INT. AE>

<INT. AE> :: = <AO><TERM>|<AO><NUM>|<AO><TERM><INT. AE>
<AO><NUM><TERM><INT. AE>

<TERM> :: = <VAR><EXP>|<VAR><TERM>

<EXP> :: =|<NUM>|Null

With the above definitions the system would be able to accept an input as:

Max X + X↑2 - 2 X YZ + 15 - XZ ↑ 2W

Lexical analysis would transform the above definition to the following UST definitions:

<AE> :: = Sign - <TERM>| Sign - <TERM><INT. AE>|
Sign - Num | Sign - Num - <TERM><INT. AE>

<INT. AE> :: = AO - <TERM>| AO - Num|
AO - <TERM>| AO - Num - <TERM><INT. AE>|
AO - <TERM><INT. AE>

<TERM> :: = Var - ↑ - Num | Var - <TERM>

The above translates to the following state graph of Figure 17.

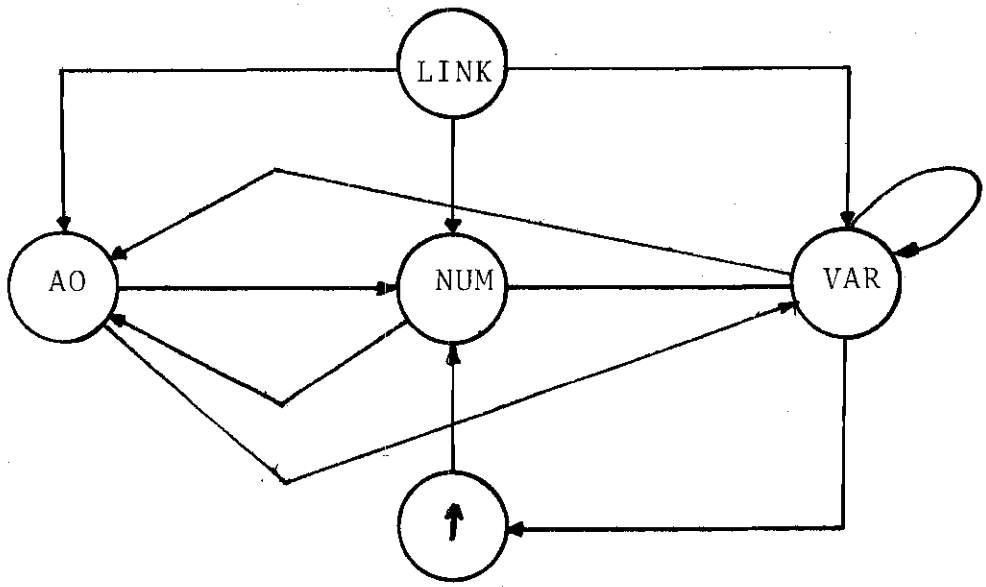


Figure 17. The State Graph of a Non-Linear Arithmetic Expression

Non-linear programming, with the exceptions of some special forms as quadratic programming, is based on function evaluation and search methods rather than on a closed form matrix driven methodology. The most efficient way of handling this situation is to simply append the non-linear expressions to a program written in a high level general purpose language. The associated compiler will check the syntax of the input. So, the syntax of a high level well known language would be the primary part of the syntax of the interfacing system. This implies that the potential user would, very possibly, be required to become familiar only with a few key words of the system. The above mentioned approach has been applied to the interface system of NLP [39], a non-linear programming package developed in Georgia Institute of Technology in 1975. NLP accepts the non-linear

expressions in Fortran syntax. The expressions are inputted in a reserved space of a subroutine. The subroutine is compiled and if it is correct it is attached to the object code of the optimization program. If it is not correct, the control is passed back to the user to edit the model. This approach has been well accepted by the students of Georgia Tech.

CHAPTER VIII

CONCLUSIONS AND RECOMMENDATIONS

VIII.1. Conclusions

From this research it was concluded that a balanced cooperation between man and computer would be beneficial to the OR community. A fundamental component of this cooperation is the communication. An efficient man-machine communication in the OR environment would substantially contribute to a balanced cooperation.

A computer-aided problem solving process is decomposed into the problem input and the optimization procedure. Usually, the initial analytical formulation must be transformed into a computer readable format. Hence, there exists a non-natural gap between the two formulations. The goal of this thesis was to narrow or eliminate this gap. To accomplish this objective a closed form methodology was presented, and its effectiveness and applicability was demonstrated.

The theory of languages and grammars represented a rigorous mathematical framework for understanding the complexity of the communication problem and finding cost effective solutions to it. The theory of automata is the corresponding framework for the design of algorithms for language processing. The power and abstraction of the above theories make them a necessity for man-machine communication

methodology.

Elements of these theories were presented in this thesis. The Chomsky classification system of grammars was presented and its association to automaton classes was demonstrated.

According to this system a grammar can be classified into a Chomsky category to which corresponds a specific computing device. A closed-form definition and description of operation of each class of automata were given, and a framework for quantifying for their cost of operations was presented. This leads to the conclusion that for a given grammar, the cost of recognizing the language generated by the grammar can be determined by observation. By observation the designer can make cost estimates of various grammar transformations.

The linearity and non-linearity of grammar definitions lead to a natural decomposition of the recognition process into lexical (linear) and syntactical (non-linear) analysis. From the theoretical framework the methodology framework was constructed.

With respect to the recognition algorithm, two approaches were presented, an abstract functional approach of automata and a state-graph approach. Advantages and disadvantages of these two approaches were discussed. By examples, it was shown that an OR language can be described by a grammar in a metalanguage and that this grammatical definition can be the

basis of an effective recognition process. Certain important components of the recognition process, such as error recovery were identified and discussed. The discussion of the methodology concentrated on ways to increase the computer share in the formulation process. Attributes as input flexibility were given special attention. Certain advanced concepts as the syntax directed interfacing were presented and their potentiality was discussed.

User psychology, an increasingly important factor of the design of man-machine systems, was introduced and special attention was brought in the long run effect of certain psychological considerations. An economic model was presented to measure the long run effectiveness. This model uses as a key factor the rate of utility of the system and demonstrates the effect of various cost and benefit considerations.

Linear programming was the main vehicle of demonstrating the effectiveness of the above concepts. The concepts about the methodology were applied to an LP package (EZLP). It was shown that the gap between the formulation process and the computer input can be practically eliminated.

VIII.2. Topics for Further Research

Research in interfacing systems for mathematical programming should be channeled in the two directions. The first direction is the improvement of the rate of communication. This attribute of the interfacing system directly

affects several other attributes as the operational cost and number of potential users, Research improving the performance of the system with respect to these attributes is taking place in relevant fields (i.e. computer sicence), in fundamentals areas as compiler theory or even artificial intelligence. In this thesis certain areas were identified which would substantially affect the rate of communication. That is to say that the system is quite sensitive with respect to these areas, which are the following:

i. Error recovery. This was discussed in Chapter IV. A good error recovery will minimize the number of runs per problem. The state of the art in this area is an ad-hoc methodology formed by a collection of ad-hoc procedures used in common cases. A closed form theory is definitely needed in this area.

ii. Syntax directed interfacing. This is the concept presented in Chapter IV for dynamically adjusting the syntactical state graph of the input language. It provides the powerful facility of user control over the lexical, syntactical and samantical analysis. The user specifies the type of the problem and the system is adjusted to accept only this type of input language.

Research in the latter area might provide the foundation for the "ultimate" interfacing system, i.e., the system which, if given a mathematical formulation, determines the type of the problem. This could be called the inverse syntax

directed interfacing. The system should possess a library of types of problems with an appropriate syntactical description of their characteristics. However, this could only be accomplished if it was supported by a rigorous mathematical theory of error recovery. Syntax directed interfacing could be viewed as an extension of the semantical analysis. Research in this area should be directed within a formal framework, such as automata, to obtain insight of the problem.

The second direction is improvement with respect to the level of communication. Research should attempt to raise the level of communication closer to the user's thinking level, i.e., closer to the problem environment. This could be accomplished by explicit or implicit facilities of representing physical situations and objectives. Research in this area should be distributed over the following areas:

i. Semantical analysis of mathematical expressions. Certain restrictions, objectives or special characteristics, sometimes can be included in the mathematical formulation as structural properties, i.e. certain amounts of valuable information can be implicitly included in the model. The information capacity from this respect, of mathematical expressions could probably be expanded if there was a methodology to relieve it. This could be accomplished by an efficient semantical analysis.

ii. Symbolic input of higher grammatical order. There

should be provisions in the system to accept "functional symbols" which could collectively represent certain knowledge or thinking approach. With a real life symbolic input the user would be able to concentrate more in the problem environment. Also, symbolic input might provide the user with a quicker "grasp" of the model he created, by using symbols he is using in analytical work on paper. For instance a large linear model would be faster visualized by the OR analyst in terms of summation symbols. Functional symbols should be devised to represent physical situations as a whole, in a particular environment. New concepts must be introduced in this area.

iii. Retrieval of human experience. This is a potential area since it is well known in the OR community and in the problem solving environment in general that human intuition and experience, in certain cases, might find better solutions than a sophisticated optimization procedure. In that respect the system should have provisions of exercising human judgement.

BIBLIOGRAPHY

1. Aho, Hopecroft, Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley.

2. A. V. Aho and J. D. Ullman, The Theory of Parsing, Translation and Compiling, V.1--Parsing, Prentice-Hall, Inc., 1972.

3. F. L. Epauer, J. Eickel, Editors, Compiler Construction, Springer Verlag, Second Edition, 1976.

4. Bazaraa, M. J. and J. J. Jarvis, Linear Programming and Network Flows, School of Industrial and Systems Engineering, Georgia Institute of Technology, 1974.

5. A. T. Berztiss, Data Structures Theory and Practice, Academic Press, 1971.

6. J. D. Beattie, Natural Language Processing by Computer, Man-Machine Studies (1969), 1, pp. 311-329.

7. Henry J. Bowlden, A List-Type Storage Technique for Alphameric Information, Comm. Acm, Vol. G, Number 3, August 1963.

8. P. C. Brillinger, and D. J. Cohen, Introduction to Data Structures and Non-Numeric Computation, Prentice-Hall, Inc., 1972.

9. British Computer Society Conference, High Level Programming Languages--The Way Ahead, September 1973.

10. T. L. Bouth, Sequential Machines and Automata Theory, J. Wiley and Sons, Inc., 1967.

11. Camsco: Report to Industry, Richardson, Texas, 1976.

12. A. Chapanis, Interactive Human Communication, Scientific American, Feb. 1975, pp. 36-49.

13. C. Cohen, A Guide to MPOS Version 2, Multi Purpose Optimization System, Vogelback Computing Center, Northwestern University, November 1975.

14. H. S. Coff, Cut Scheduling for Optimum Fabric Utilization in Apparel Production, M.S. Thesis, School of Textile Engineering, Georgia Institute of Technology, November, 1976.

15. Melvin E. Conway, Design of a Separate Transition-Diagram Compiler, Comm. Acm., Vol. 6, Number 7, July 1963.

16. C. Donaghey, P. Sewan, D. Singh, A Beginner's Language for LP, Industrial Engineering, December 1970, p. 17.

17. J. J. Donovan, Systems Programming, McGraw-Hill, 1972.

18. Jay Earley, An Efficient Context-Free Parsing Algorithm, Comm. Acm., Vol. 13, Number 2, Feb. 1970.

19. J. R. Emshoff and R. L. Sisson, Design and Use of Computer Simulation Models, MacMillan Publishing Co., 1972.

20. S. J. Fenves, Problem-Oriented Languages for Man-Machine Communication in Engineering, IBM Scientific Computing Symposium on Man-Machine Communication, IBM, 1966, p. 43.

21. J. M. Foster, Automatic Syntactic Analysis, MacDonald and American Elsevier, Inc., 1970.

22. O. N. Garcia and A. B. Marcovitz, A Syntactic Approach to Teaching Computer Languages, Engineering Education, Jan. 1969, pp. 426-427.

23. Tom Gilb, Software Metrics, Winthrod Computer Series, 1977.

24. M. Gross and A. Lentin, Introduction to Formal Grammars, Spring & Verlga, 1967.

25. A. M. Horman, A Man-Machine Synergistic Approach to Planning and Creative Problem Solving, Part 1 and Part 2, Int. J. Man-Machine Studies (1971)3, pp. 167-184 and pp. 241-267.

26. E. Horowitz, Editor, Practical Strategies for Developing Large Software Systems, Addison-Wesley, 1975.

27. E. T. Irons, An Error-Correcting Parse Algorithm, Comm. Acm., Vol. 6, Number 11, November 1963.

28. J. J. Jarvis, F. H. Cullen, C. Papacostadopoulos, EZLP: An Interactive Computer Program for Solving Linear Programming Problems, School of Industrial Engineering, Georgia Institute of Technology, September 1976.

29. L. R. Johnson, System Structure in Data, Programs and Computers, Prentice Hall, Inc., 1970.

30. H. Katzan, Jr., Advanced Programming, Van Nostdanb Reinhold Co., 1970.

31. Peter G. W. Keen, Interactive Computer Systems for Managers: A Modest Proposal, Sloan Management Review, Fall 1970, pp. 1-17.

32. M. Konopasek, C. Papacostadopoulos, H. Coff, Application of Linear Programming Techniques in Cut Scheduling for Better Fabric Utilization, School of Textile Engineering, Georgia Institute of Technology, February 1977.

33. J. A. N. Lee, The Anatomy of a Compiler, Van Nostrand Reinhold Company, 1974.

34. J. Martin, Design of Man-Computer Dialogues, Prentice-Hall, Inc., 1973.

35. H. L. Morgan, Spelling Correction in Systems Programs, Comm. Acm., Vol. 13, Number 2, February 1970.

36. Robert McNaughton, The Theory of Automata, Survey Advances in Computers, pp. 379-421.

37. A. G. Oettinger, Linguistic Problems of Man-Computer Interaction, IBM Scientific Computing Symposium on Man-Machine Communication, IBM, 1966, p. 33.

38. W. D. Orr, Conversational Computers, John Wiley & Sons, Inc., 1968.

39. C. Papacostadopoulos, NLP-User Documentation, School of Industrial and Systems Engineering, Georgia Institute of Technology, September, 1975.

40. L. Press, Toward Balanced Man-Machine Systems, Int. J. Man-Machine Studies (1971) 3, pp. 61-73.

41. Proceedings of the IBM Scientific Computing Symposium on Man-Machine Communication, IBM, 1966.

42. P. Reisner, Use of Psychological Experimentation as an Aid to Development of a Quary Language, IEEE, Software Engineering, Vol. SE-3, No. 3, May 1977.

43. C. J. Routhaan, Solving Problems with Long Run Time, IBM Scientific Computing Symposium on Man-Machine Communication, IBM, 1966, p. 3.

44. H. Sackman, Computers System Science and Evolving Society, J. Wiley & Sons, Inc., 1967.

45. W. D. Seider, Time Sharing in Engineering Education, Engineering Education, January 1969, pp. 384-386.

46. J. C. Shaw, JOSS: Experience with an Experimental Computing Service for Users at Remote Typewriter Console, IBM Scientific Computing Symposium on Man-Machine Communication, IBM, 1966.

47. H. G. Thuesen, W. T. Fabrycky, and G. J. Thuesen, Engineering Economy, Prentice Hall, 1977.

48. J. T. Tov, Editor, Information Systems, Coins IV, Plenum Press, 1974.

49. S. Tseu, Interactive Command Language Design Based on Required Mental Work, Int. J. Man-Machine Studies (1975) 7, pp. 135-149.

50. University of Illinois, Manual of Computer Programs and Statistical Analysis, Statistical Service Unit, University of Illinois, Urbana, Illinois, 1964.

51. G. R. Wagner, M. M. McCants, Conversational Linear Programming for Experimental Learning, Engineering Education, Vol. 62, No. 7, April 1972.

52. R. Wall, Introduction to Mathematical Linguistics, Prentice-Hall, Inc., 1972.

53. F. W. Weingarten, Translation of Computer Languages, Holden-Day, Inc., 1973.

54. D. B. Young, Benefit Profile Analysis in Environmental Decision Making, PhD Dissertation, The University of Texas at Austin, 1970.

55. D. Young, Some Useful Continuous Discounted Cash Flow Formulas (1972), School of Industrial and Systems Engineering, Georgia Institute of Technology.

56. D. B. Young, Personal communication, School of Industrial and Systems Engineering, Georgia Institute of Technology, June 1977.